Run JAVA Code in C/C++!

C/C++ Users Journal

September
1996
Vol. 14, No. 9

C/C++ Users Journal™

Advanced Solutions for C/C++ Programmers

STL

- Cleaner Code with STL's *map*
- Two Important STL Books Compared
- STL Help on the Web

Simple-as-a-Stopwatch Timing Class
Finding the Right Screen Library
Keep Floating-Point Sums Precise

P. J. Plauger
STL's Treasure Trove of Algorithms

Dan Saks
Making C++ Grammar More Intuitive

Pete Becker
When to Break the Rules of OOP

Bobby Schmidt
The Winding Road from C to C++

$4.95 U.S. Canada $5.95
09 >
0 74470 74098 8
A Miller Freeman Publication

Communications?

There's lots of

Data Compression?

New Stuff

From Greenleaf

http://www.gleaf.com/~gleaf

visit us today...

Greenleaf Comm++ 3.0

Async communications the C++ way. Classes for file transfer, terminal emulation, screen drivers, modem and port control, flow control. Supports 16550, smart and non-smart multi-port boards. Supports Windows 95, NT, Windows 3.1x, OS/2 Warp, Novell NASI and 32-bit DOS.

NEW—ViewComm for Windows

Asynch communications debugger for Windows 95 and NT. Peek at RS232 lines, see data and control signals in realtime or review captured data. Flexible trigger system provides start-stop control of display or capture. Graphical display of modem & control signals. Data display in hex, octal, decimal, binary. ASCII & EBCDIC data. Character or hex displays. Searches. Many extras. ViewComm/DOS also available.

CommLib 5.2 Level 2

Asynch communications for Visual Basic, C, and C++. Supports all popular smart & non-smart multi-port boards, 16550, networked ports. One function call modification to change hardware type. File transfers, modem and port controls, flow controls, terminal emulations, plus hundreds of extras. Supports Windows 95, NT, Windows 3.1x, and 16- and 32-bit DOS. CommLib--the industry standard for 12 years.

ArchiveLib 2.0

Greenleaf proprietary and PKZIP 2.0x compatible compression engines. Visual Basic, Pascal, Delphi, and C/C++ APIs. Flexible archiving including compatibility with PKZIP. Archive to memory or file. Supports Windows 95, NT, Windows 3.1x, OS/2 Warp, and 16- and 32-bit DOS.

No royalties. Lots of examples. Onscreen documentation. Library products come with free source code. 30-day no-questions money-back. Download free test-drive software.

☎ *Call today for complete information, demo, or to order. MasterCard, VISA, AmEx and Novus; approved PO's*

1-800-523-9830

GREENLEAF Software©

Greenleaf Software, Inc.
16479 Dallas Parkway, Suite 570
Dallas, TX 75248
800.523.9830 214.248.2561
FAX 214.248.7830
BBS 214.250.3778 28.8/8/N/1

http://www.gleaf.com/~gleaf
email info@gleaf.com

❑ Request Reader Service #100 ❑

New Version! 6.0

HIGH PERFORMANCE

CONVERSION

SCALE-TO-GRAY

IMAGE PROCESSING

THUMBNAILS

PRINTING

SCANNING

FAST DISPLAY

COMPRESSION

and more!

The Best Imaging Toolkit - Is Now Even Better!

AccuSoft Corporation announces a totally new product, **ImageGear**™, the next generation in imaging technology. ImageGear, the new version of the AccuSoft Image Format Library™, is more than just an upgrade, it is a **whole new level of the quality and performance** that has made AccuSoft the leading supplier of imaging toolkits. Just for starters, we redesigned all the image filters to add several new features to ensure that our support for image formats is by far the most comprehensive. We even added **file-info executables** for every format we support - and, of course, we added **several new formats.**

ImageGear has a **new and improved API** that makes integration even easier, plus many new features including totally redesigned display functions for **faster image display**, sub-pixel accuracy, transparency, and an automatic image window with many built-in features.

We also added new **GUI functions** including a comprehensive thumbnail browser, pan and zoom windows, **magnifying glass**, page sorter and more. All with **complete programmability**. We completely revamped the image processing to add **dozens** of new functions and better performance. In addition, we have improved the **scanning**, image compression, display effects, and just about everything else!

So get into **high gear** with ImageGear and remember - the best imaging toolkit makes the best applications - **ImageGear 6.0!**

IMAGEGEAR™

30 MINUTE Delivery™

You can start using AccuSoft products today by taking advantage of our unique 30 Minute Delivery Program.

Toll Free: (800) 525-3577
Website: accusoft.com

AccuSoft
High Performance Imaging

Two Westborough Business Park Westborough, MA 01581 Tel (508) 898-2770 FAX (508) 898-9662

Versions Available

Windows
Win95\NT
MIPS NT
Alpha NT
VBX
OCX
OS/2
SUN OS
Solaris
HP-UX
RS/6000
SGI
SCO
MAC
PowerMac

Over 45 File Formats

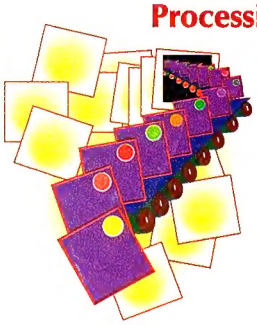
JPEG
TIFF
Group III
Group IV
PCX
TGA
PGM
DIB
IMT
DCX
BMP
KFX
RLE
LV
CALS
ATT
CLP
XWD
XBM
CIF
IFF
SUN
PNM
IOCA
GX2
XPM
ASCII
CUT
GEM
BRK
MAC
PSD
MSP
PNG
PCD
IMG
IGF
SGI
ICO
PBM
PPM
MO:DCA
WMF[1]
WPG[1]
PICT[1]
EPS[1]
GIF[2]
ABIC[3]
JBIG[3]
DICOM[3]
and others

ACCUSOFT IMAGE™ GUARANTEE

C/C++ Users Journal™
Advanced Solutions for C/C++ Programmers

September 1996
Vol. 14, No. 09

STL



FEATURES

Cover illustration by Twyla Watson Bogaard.

C/C++ Users Journal (ISSN 1075-2838) is published monthly by Miller Freeman, Inc., 600 Harrison Street, San Francisco, CA 94107 (913) 841-1631. Periodicals paid at San Francisco, CA and additional mailing offices. **POSTMASTER:** Send address changes to C/C++ USERS JOURNAL, P.O. Box 52582, Boulder, CO 80322-2582. **Subscriptions:** Annual renewable subscriptions to C/C++ Users Journal are $34.95 US, $46 Canada and Mexico, $65 overseas. Payments must be made in US dollars. Make checks payable to C/C++ Users Journal. **GST (Canada):L #129065819**

COLUMNS

DEPARTMENTS

CUJ Online Source Code

Obtain all code published (and some unpublished) in CUJ from:

FTP: ftp.mfi.com in pub/cuj
BBS: The Courts of Chaos, 501-985-0059; EmmaSoft Shareware Board, 607-533-7072; Phoenix Chapter ACM Library, 602-821-1162; C_BBS (The Netherlands), +31-(0)-4930-23061 or +31-(0)-4930-20792.
CompuServe: GO SDFORUM, Library 7 (R&D Publications)
GEnie: In the IBM-PC Roundtable at page 615 (Keyword: IBMPC).
Monthly Code Disk: Call Miller Freeman, Inc., Customer Relations, 1-800-444-4881 or 913-838-7500.

C/C++ Users Journal™
Advanced Solutions for C/C++ Programmers

EDITORIAL

Senior Editor — P. J. Plauger
Managing Editor — Marc Briand
Contributing Editors — Dan Saks, Pete Becker, Bobby Schmidt, Victor R. Volkman
Consulting Editor — Chuck Allison
Associate Publisher — Martha Masinton
Publishing Director — Peter Hutchinson

PRODUCTION

Art Director — Twyla Watson Bogaard
Production Editor — Amy Pettle
Advertising Materials — Lori White

ADVERTISING AND MARKETING

Director of Sales & Marketing — Bill Uhler
Acct. Manager, East/United Kingdom — Ed Day, 913-838-7547, eday@mfi.com
Acct. Manager, Midwest — Christine Woodley, 913-838-7546, cwoodley@mfi.com
Acct. Manager, West — Julie Thornton, 913-838-7541, jthornto@mfi.com
European Advertising Representative — breakout! marketing, Duevelsbeker Weg 4, D-24105 Kiel, Germany, +49 431-801740, FAX +49 431-801797, 100332.1704@compuserve.com
Sales Support Manager — Joni Hernly
Circulation Manager — Cherilyn Olmsted

CUSTOMER SERVICE

1-800-365-1364
303-678-0439
FAX 303-661-1885

ABC AUDITED

Advertising: For rate cards or other information on placing advertising in C/C++ Users Journal, contact the advertising department at (913) 841-1631, or write C/C++ Users Journal, 1601 W. 23rd St., Ste. 200, Lawrence, KS 66046-2700.
Customer Service: For subscription orders and address changes, contact C/C++ Users Journal, P.O. Box 52582, Boulder, CO 80322-2582; Telephone 1-800-365-1364 or +1-303-678-0439; FAX +1-303-661-1885; e-mail cujsub@mfi.com.
Back Issues and Article Reprints: 1-800-444-4881 or +1-913-841-1631.

MILLER FREEMAN INC.

Chairman/CEO — Marshall W. Freeman
President/COO — Thomas L. Kemp
Senior VP/CFO — Warren "Andy" Ambrose
Senior Vice Presidents — David Nussbaum, Darrell Denny, Donald A. Pazour, Wini D. Ragus
Vice President/Software — Regina Starr Ridley
Vice President/Productions — Andrew A. Mickus
Vice President/Circulation — Jerry Okabe

Miller Freeman
A United News & Media publication

Phar Lap's ETS™ TCP/IP allows your mainframes, workstations, or PCs to communicate with products on the factory floor, in the lab, or at a remote field site.

ETS TCP/IP Software—Harness the Power

The ETS TCP/IP software is part of Phar Lap's TNT Embedded ToolSuite®, Realtime Edition version 8.5 SDK, complete with Realtime ETS Kernel. It is based on a subset of industry standard Win32® and WinSock APIs, the same APIs as Windows® NT® and Windows 95—making it easier and faster to get your product to market!

Ethernet Support—Stable as Ever

Software drivers and sources for:

- 3COM: 3C509
- SMC: WD803, WD8216
- Novell: NE2000 compatibles

For systems with custom interfaces or unsupported cards, the TNT ToolSuite has solutions to make it easy to create your own.

Serial/Dial-up Support—Lots of Horsepower

When designing remote monitoring systems, your system can be connected to a telephone in order to dial up a host computer directly or via an Internet Service Provider (ISP). Either way, it saves you money!

ETS Web Connection—Ahead of the Pack

Phar Lap provides all of the required Web technology including an HTTP server (Web server) and an HTML on-the-fly library for converting raw data into HTML pages for the Web or, if you prefer, just part of your customer's Intranet.

Network Protocols Supported—Putting You on Track

TCP	IP	UDP
ICMP	ARP	RARP
DNS	HTTP	FTP
TELENET	HTML	

As you can see, Phar Lap's TNT Embedded ToolSuite, Realtime Edition version 8.5 SDK has it all: development tools, a Realtime ETS Kernel, and ETS TCP/IP. So stop horsing around and visit us at the "World's Smallest Web Server" URL http://smallest.pharlap.com or call a Phar Lap sales representative today!

(617) 661-1510

Embedded Development—Simply on Target™

Phar Lap Software, Inc. 60 Aberdeen Avenue, Cambridge, MA 02138 Tel: (617) 661-1510 Fax: (617) 876-2972

❑ Request Reader Service #104 ❑

Looking For Version Control? See What The Experts Have Found.

4½ "Source Integrity is more than just a file management system. It performs revision control and simplifies data retrieval, and groups files as projects... so stop reading and get back to writing code. Only this time, take Source Integrity along; it'll make the ride a lot easier." (Thom Duncan—LAN TIMES)

RATING: ★★★★ "One of my favorite features is event triggers... Source Integrity's implementation of this feature is especially great because it lets you build a code block in a simple but powerful scripting language." (SOFTWARE DEVELOPMENT MAGAZINE)

"Would you like to know how SI (Source Integrity) integrates with Microsoft Corp. C++ and Borland International Inc. C++? One word: seamlessly." (Thom Duncan—LAN TIMES)

"One highlight of the product is a visual merge facility that allows developers to merge different versions with the click of a mouse. 'Sandbox' personal workspaces are supported, where developers can make and test the effects of changes independent of the project as a whole." (PC TECHNIQUES)

Finding the right version control solution is crucial to maintaining the integrity of your software. You need a proven solution that will guarantee the results you're looking for. Our award-winning software has brought us accolades from respected industry leaders worldwide, establishing *MKS Source Integrity* as the experts choice for version control management.

MKS Source Integrity handles multiple development environments including Borland C++, Borland Delphi, Microsoft Visual Basic, Microsoft Visual C++, PowerBuilder, and Watcom C++. Unlike traditional client/server SCM tools, the MKS solution encompasses distributed development environments, the entire Web team, and intranet developers. If that doesn't convince you, maybe this will: over 450,000 developers worldwide use MKS products to help them accelerate their team's productivity, protect their software assets and guarantee overall source code integrity.

BUYERS ASSURANCE SEAL INFO WORLD

SOFTWARE Development PRODUCTIVITY AWARD 1995

In the end, it's your review that matters most. To find out more about the MKS advantage, check out our Web site for a full copy of the above articles and download our interactive Source Integrity demo, or call 1-800-613-7535, ext. 697.

Member Netscape Development Partners Program.

mks MORTICE KERN SYSTEMS INC.

Mortice Kern Systems Inc.
Tel: 519-884-2251 Fax: 519-884-8861
http://www.mks.com/solutions/si/cu600/

MKS Germany: TEL +49 711 167140 MKS Nordic: TEL +45 33 25 6555 MKS UK: TEL +44 171 624 0100

Managing Change With Integrity

Editor's Forum

C/C++

Okay, folks, it's standards update time again. I'm still jet lagged from several grueling weeks of travel. In late June, I convened the X3J11/WG14 (C) meeting in Amsterdam. Got home for the week of the Fourth, which also included a quick overnight trip to Washington DC. No sooner had I readjusted to Eastern Standard Time but I had to go back to Europe for the X3J16/WG21 meeting (C++) meeting in Stockholm. At the time this schedule was set up, I didn't think it would affect me. Had I But Known, I would have used my limited powers as WG14 Convener to make it more humane.

In any event, the C meeting went smoothly, as it usually does. About 20 attendees, representing six countries, were present. Most are old friends by now and work smoothly together. The committee looks to be on schedule to have the technical particulars of C9X decided by the end of 1996. A formal standard should still occur in time to make that 9X an honest designation.

Without going into a lot of detail, I'll simply say that C9X looks to be the Standard C you've come to know and love with a number of features added primarily for numerical programming. That's largely because the Numerical C Extensions Group was for several years the principal clearing house for extensions to Standard C. The C committee is *very* conservative, turning down essentially all proposals for which there is no existing practice somewhere. The sad thing, to me, is that any hope of adding class extensions to C, a la C++, is probably dead. The principal champion for this ambitious addition, Bob Jervis at Sun Microsystems, is now caught up in the Java frenzy and can no longer attend meetings.

The C++ meeting went fairly smoothly too, at least through Thursday evening when I left. About 50 attendees, also representing six countries, were present. The goal of this meeting was to complete the changes in response to the first Committee Draft ballot, and to vote out the draft for a second such ballot. Actually, it was the goal of the *previous* meeting to do so — the March meeting in Santa Cruz couldn't quite handle all the issues in time, however.

As far as I know, the committee didn't actually vote out the draft this time. A few issues remain to be handled. More important, several national bodies wanted time to review the updated draft before sending it up the line to ISO. But the committee did commit to a feature freeze. If they hold to their promise to themselves, then Standard C++ may actually be pretty well nailed down by now.

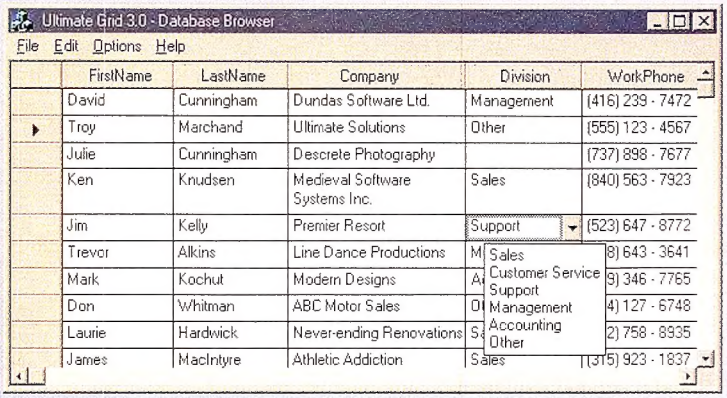
Pete Becker and Dan Saks were also at the meeting, so you should be reading more detail in these pages, from various perspectives, in the coming months. Do stay tuned.

P.J. Plauger
Senior Editor

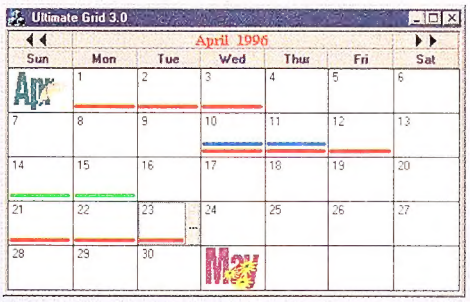
Announcing a New *CUJ* Web Service:
Our Favorite Source Code Sites
Visit our web page at `http://www.cuj.com`

Ultimate Grid™ v3.0

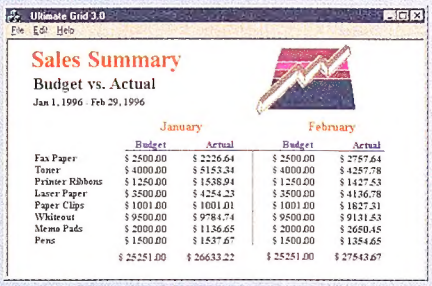
100% MFC Object Oriented Design

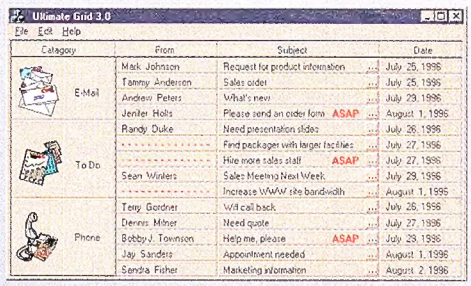


- **The Premier Grid Control for C++ development now breaks all the barriers. A new 100% MFC Object Oriented design redefines ease of use. Unparalleled power, Unparalleled features, Ultimate performance.**
- Built-in support for DAO, ODBC, and 3rd party database libraries from Codebase, Faircom and Greenleaf make database integration easier than ever.
- Connect directly to any datasource: arrays, memory mapped files, text files, proprietary databases, SQL, real-time, and more. Ultimate Grid gives you total flexibility.
- Includes a high-powered memory manager for preloading the grid, and the ability to load data virtually.
- Multiple cell types: Bitmaps, checkboxes, radio buttons, drop lists, progress bars, spin buttons, multi-line, auto font size, and more. Too many to list, download our demo for more examples!



- Use as a CView, CWnd, CDialog or dialog resource! Fully UNICODE enabled, includes Print/Preview, Find/Replace, and much, much more.
- Very simple to use: implement a DAO connected grid with full editing, in just minutes. If you wish, customize the grid's operation. You control everything down to the smallest detail.
- Features galore! Inline masked editing, OLE drag and drop, multiple text styles, hints, multiple selection, adjustable column and row height, multi-line cells, multiple data sources, cell joining, multiple fonts per cell, and more!.
- Customize the grid to your needs by easily generating your own custom cell styles!
- Includes comprehensive source code examples and demonstration code to get you started right away. Includes both a printed manual and comprehensive on-line help.
- Royalty-Free!
- Site Licensing, license your whole development team for one low price. Call for pricing.
- **** Non MFC versions available, call for full product details.**
- Pricing: $149.00 / with source $349.00





Ultimate TCP/IP™

Server and Client TCP/IP Libraries Includes 16 and 32 bit!

- A full set of C++ class libraries for developing internet client and server applications. Includes sourcecode for FTP, HTTP, DNS, FINGER, SMTP/POP3 and more!
- Build standard or custom applications in just minutes. Modify our sample server and client code to get exactly what you want instantly.
- Includes functions and examples to easily create multi-threaded internet server applications that run as true Windows NT services. Our toolkit makes it easy to exploit the latest features of WIN32.
- Royalty-Free!
- Site Licensing available.
- Pricing: $199.00 / Libraries $499.00 / Libraries with Source

Ultimate VB Service™

Turn any app into a NT service - Instantly!

- Finally, run any application as a Windows NT service. Run Visual Basic, Delphi, MFC, C++, C, Fortran, Pascal, and anything else you might want, even DOS apps!
- Well behaved, full featured, and adheres to the NT security model.
- Ultimate VB Service allows you to give your projects that extra power, with absolutely no hassle.
- Royalty-free!
- Pricing: from $199.00

For a complete list of available products and information

DUNDAS SOFTWARE

sales@dundas.com

Voice: (416) 239-7472

Fax: (416) 239-2183

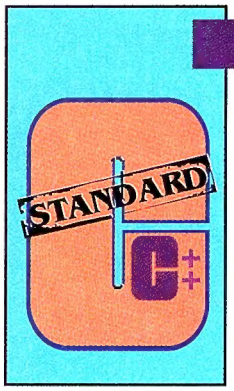
(800) 463-1492

www.dundas.com

- Prices shown in U.S. dollars.
- Call Today! Visa, Amex, MasterCard, or money order.
- We will deliver via internet, Compuserve, mail or courier.
- 30 day Money back guarantee. If you do not think these products are the best in their class, we will gladly give you your money back.

202-4800 Dundas Street West, Etobicoke, Ontario M9A 1B1 240 Portage Rd., Suite 670, Lewiston, New York, USA 14092

Request Reader Service #107



P. J. Plauger

Introduction to <algorithm>

Programmers should not have to reinvent the wheel, much less test it. STL helps you avoid both, by providing tested implementations of the most commonly needed algorithms.

Introduction

Last month, I introduced the general topic of algorithms, at least as they are provided by the Standard Template Library, or STL for short. (See "Standard C/C++: Algorithms," *CUJ,* August 1996.) An algorithm in this context is a template function that performs a widely useful function, such as searching a sequence or sorting it. Such functions are typically parametrized in terms of the types of iterators they manipulate to access or generate sequences. (See "Standard C/C++: Iterators," *CUJ,* February 1996.)

STL provides many dozens of algorithm template functions. The vast majority of them are defined in the template <algorithm>, which is by far the largest of the STL headers. (In my implementation of the draft Standard C++ library, it is more than twice as big as the next largest header. But that is partly an artifact of how I chose to chop up the extensive locale machinery.) A handful also appear in the header <numeric>, for some reason.

Even discounting near duplicates, which I discuss below, you can still count over five dozen different algorithms in STL. That's a real treasure trove. Even nicer is the fact that many of these algorithms are hard to get right. It also helps that the template functions automatically adapt when different categories of iterators can profit from different approaches.

P.J. Plauger is senior editor of *C/C++ Users Journal*. He is convener of the ISO C standards committee, WG14, and active on the C++ committee, WG21. He developed *Lib Suite ++*, the Plum-Hall validation suite for the Standard C++ library. His latest books are *The Draft Standard C++ Library*, *Programming on Purpose* (three volumes), and *Standard C* (with Jim Brodie), all published by Prentice-Hall. You can reach him at pjp@plauger.com.

I don't intend to show you every single algorithm defined by the Standard Template Library, but I do intend to show you many of them. In the process of rewriting the code distributed by Hewlett-Packard, I learned a lot about working with iterators in particular and with templates in general. I think you will find many coding techniques worth emulating here.

This is the first of three installments that describe the algorithms of STL. It is also the most eclectic, I believe. What you see this month is a number of fairly small template functions, each of which does a simple job simply and well. The goal, in each case, is to generate code as good as or better than a good programmer can craft by hand for a given specialization. That's what it takes to make a library that is truly reused, not just nominally reusable.

Non-Sequence Functions

Nearly all the STL algorithms manipulate sequences using iterators. A handful, however, do not. For example, Listing 1 shows two versions of the template function max, which returns the larger of its two arguments. Actually, only the first of the two versions is guaranteed to do so. It uses operator< to compare the two arguments and determine which is larger.

The second version accepts a *function object* argument as well. As I described last month, such a creature is either a function pointer or an object of some class that overloads operator(). In either case, the template function must be able to name the object in a function call expression, with some specified signature. In this particular case, the second form of max replaces x < y with pr(x, y), where pr is the function object.

If you call max with the function object greater<T, T> (defined in the header <functional>), you get the same behavior as when you omit the third argument. But you can trot in any sort of function object in its place, so long as it accepts a signature of the proper form. Using less<T, T> (also defined in <functional>), for example, works just fine, but makes a lie out of the name max.

Nearly all the algorithm template functions come in two versions. One is hard wired for a widely used predicate, such as operator<. The other accepts an additional argument, whose type is a template parameter, so you can supply the predicate in the form of a broad range of function objects. Given the current state of optimization, the duplication of code is often a good idea.

I won't show the other non-sequence template functions, since their implementation is obvious enough given the example in Listing 1. I'll simply mention that they are min, with and without a predicate argument, swap, and iter_swap. Only the last name is less than obvious — it swaps two items given iterators that designate them.

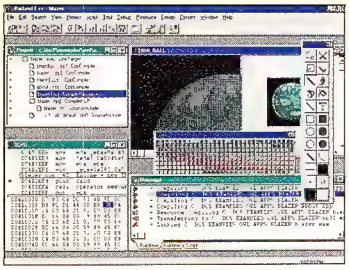
Searching

Listing 2 shows max_element, the extension of max for a sequence with an arbitrary number of elements. Note that it returns an iterator designating the largest element. In the degenerate case where the sequence is empty (first == last), it returns an iterator just past the end of the empty sequence.

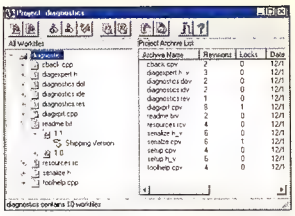
Note also that max_element comes in two versions, one taking a predicate argument, just like its simpler cousin in Listing 1. From now on, I won't show the more general form. There's quite enough code to look at without the added redundancy. I also won't show the obvious companions, template function min_element, with and without a predicate argument.

Sometimes you want to consider a sequence as an ordered group of related elements, much like the characters in a string literal such as "abc". Listing 3 shows template function lexicographical_compare. It compares two sequences element by element to test whether the first is ordered "before" the second, by the usual rules for ordering words by their spelling. As usual, you can supply a predicate argument to give your own meaning to "before" for two elements (version not shown).

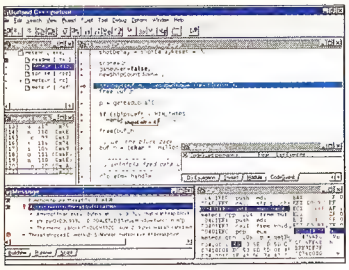
Listing 3 also shows mismatch, which compares elements from two sequences until it finds a pair that don't compare equal. It then returns an object of class pair



Borland C++ 5.0
Take advantage of the most productive C and C++ environment (included).

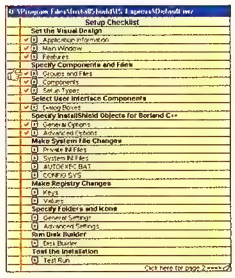


PVCS
Manage your source code easily and safely with powerful PVCS version control (included).



CodeGuard
Boost application quality with automatic bug detection and diagnosis (included).

Internet programming with Java
Get the best integrated Java™ development environment including a GUI debugger written in Java. Plus, get the App*Accelerator*™ to boost Java application and applet performance by up to ten times (included).*



InstallShield Express
Build your installation with point-and-click ease (included).

Cut development time using the New Borland C++ Development Suite 5.0

The only comprehensive, integrated suite available. The new Borland® C++ Development Suite 5.0 was designed to speed your development. It is the only product to bring together four essential phases of the development process: coding, testing, version control, and install creation—all totally *integrated* for 32-bit Windows development. Plus, get integrated Java tools and App*Accelerator*.

Borland C++ 5.0—the most productivity on Windows 95/NT. The suite includes the complete Borland C++ 5.0, designed to bring state-of-the-art innovations in productivity to Windows 95/NT development. Borland C++ 5.0 features a new 32-bit hosted IDE, parallel 32- and 16-bit development from a single IDE, updated C++ support, updated OWL 5.0 (with more than 150,000 lines of pretested reusable code), ObjectScripting, and MFC library compilation support.

Powerful new ObjectScripting saves you time by automating your build process, integrating tools and utilities, and even adding new features and experts to facilitate compliance with corporate, team, or individual standards. With ObjectScripting, the only limit to productivity is your imagination.

CodeGuard 32/16 automatic bug detection. Use CodeGuard™ to detect, pinpoint, and diagnose elusive memory and resource bugs in your 32- and 16-bit Windows applications without changing a single line of code. And by double-clicking on your error, CodeGuard will automatically take you to the offending line of code.

Seamless PVCS Version Manager. Now PVCS version control comes built into your development environment. Check-in code, check-out code, retrieve old versions, visually compare changes in full color, and protect all of your work. Great for both solo and team-based code management.

InstallShield® Express point-and-click install creation. Use any of 13 prebuilt dialog boxes. Visually build disks, test installation, add uninstall functionality, and create a distribution master.

Find out how much more productive you can be with a suite.

Compare the Facts

Features supported	Borland C++ Development Suite 5.0	Borland C++ 5.0	Visual C++ 4.0
Complete support for Windows 95/NT	✔	✔	✔
Parallel 32-bit and 16-bit development	✔	✔	—
ObjectScripting (fully programmable IDE)	✔	✔	—
Visual database development	✔	✔	—
OCX container support	✔	✔	✔
VBX support for both 32- and 16-bit applications	✔	✔	—
FREE Java tools for the Internet, including GUI debugger	✔	✔	—
Windows 95 controls supported under Windows 3.1	✔	✔	—
MFC compilation support	✔	✔	✔
ANSI/ISO Standard C++ Library, including STL	✔	✔	—
Integrated 32-bit debugging and resource editing	✔	✔	✔
Point-and-click install expert with prebuilt components	✔	—	—
Automatic uninstall support for Windows 95	✔	—	—
Detects bad pointers, even outside of API calls	✔	—	—
Automatic detection, location, and diagnosis of memory bugs	✔	—	—
Full color visual differencing of source code versions	✔	—	—
Label and maintain source files for beta, release, etc. versions	✔	—	—
App*Accelerator* for up to ten times faster Java code	✔	—	—

Estimated Retail Price
$349†

Estimated Retail Price
$249†

Get FREE update to Borland C++ 5.0– now includes MFC.
PLUS, create your own Web site with free DeltaPoint Quicksite.

Check out Borland Online to find out more about MFC and new BC++ Development Suite with Design Tools 5.0.

http://www.borland.com

Or for more information call
1-800-336-6464, ext. 50599
CompuServe: GO BORLAND ■ Canada: 1-800-461-3327

Borland
Making Development Easier

❑ Request Reader Service #108 ❑

Egghead Software 1-800-555-5555
CompUSA 1-800-COMPUSA
Computer City 1-800-THECITY
Programmers' Shop 1-800-421-8006

*Free for a limited time. †Competitive product owner's upgrade price. Java is a trademark of Sun Microsystems, Inc., and refers to Sun's Java programming language.

that stores the two offending iterators. Template function `equal` uses `mismatch` to test whether the two sequences are completely equivalent. Both functions have a version that takes a predicate argument. In this case, however, the predicate replaces `operator==`, not `operator<`.

Another slew of algorithms searches a sequence for various conditions. Listing 4 shows a representative sampler. Template function `find` finds the first element that compares equal to a value you specify. You can replace the value argument with a predicate object, but you must then call the function `find_if`. Template parameter matching is not smart enough always to safely distinguish between a value and a function object.

`adjacent_find` finds two adjacent elements that compare equal, or that satisfy a predicate that you specify with an added argument. Note that this function requires forward iterators. You can't use an input iterator to designate any but the latest, unconsumed element in a sequence.

`count` counts all the elements that compare equal to a value you specify. As with `find`, the version that takes a predicate argument in place of a comparison value is called `count_if`. The return type of `count` has recently been changed to be the type associated with its iterator parameters. That change requires partial specialization of templates, however, which is currently unavailable in commercial compilers. So I replace the proper return type with `size_t`, which is usually close enough.

`search` searches for a subsequence within a sequence. Note the standard trick of calling yet another template function (`_Search`) to determine the types used to store distances between the different iterator types. Once partial specialization becomes widespread, the use of `_Dist_type` can be discontinued in favor of the notation required for the return type of `count`. As usual, you can replace the equality comparison for pairs of elements by supplying an added predicate argument. `search_n` behaves much the same, except that the subsequence to search for is a repetition of `n` copies of a value you specify.

`find_end` is like `find`, except that it returns the last match in the sequence, not the first. And `find_first_of` is similar to `find`, except that it returns the first element that matches any of the elements in a second sequence. Both functions accept the usual added predicate argument.

Listing 1: Template function max

```
// TEMPLATE FUNCTION max
template<class T> inline
   const T& max(const T& x, const T& y)
   {return (x < y ? y : x); }

// TEMPLATE FUNCTION max WITH PRED
template<class T, class Pred> inline
   const T& max(const T& x, const T& y,
                Pred pr)
   {return (pr(x, y) ? y : x); }
//End of File
```

Listing 2: Template function max_element

```
// TEMPLATE FUNCTION max_element
template<class FwdIt> inline
    FwdIt max_element(FwdIt first,
                      FwdIt last)
    {FwdIt x = first;
    if (first != last)
        for (; ++first != last; )
            if (*x < *first)
                x = first;
    return (x); }

// TEMPLATE FUNCTION max_element WITH PRED
template<class FwdIt, class Pred> inline
    FwdIt max_element(FwdIt first,
                      FwdIt last, Pred pr)
    {FwdIt x = first;
    if (first != last)
        for (; ++first != last; )
            if (pr(*x, *first))
                x = first;
    return (x); }
//End of File
```

Generators

A handful of template functions perform a specified operation for each element of a sequence. Listing 5 shows these creatures. `for_each` calls the function object you specify for each element of the sequence. Note that it does nothing with any return value. By contrast, `generate` stores the return value of the function object you specify in each element of a sequence, but it lets you specify no argument to the function. `generate_n` does much the same thing for the first `n` elements of a sequence.

`transform` puts the two services together. The unary version applies the function object you specify to each element of one sequence to determine the return value to store in another sequence. The binary version calls the function with two arguments — elements from two sequences — to generate a third sequence.

A simpler way to generate a sequence is to copy an existing one. Listing 5 shows an assortment of such template functions. `copy` copies a specified sequence from beginning to end, while `copy_backward` copies from end to beginning. Knowing which way a copy proceeds is invaluable when you have to copy between areas that overlap.

`fill` replicates a value you specify throughout a sequence. `fill_n`, on the other hand, copies the value you specify to the first `n` elements of a sequence. Note the difference in iterator categories, once again.

Finally, `swap_ranges` exchanges two sequences. It uses the template function `iter_swap` that I showed much earlier. (Strictly speaking, this is not a sequence generator, but sometimes it's hard to characterize a group of loosely related functions.)

Listing 3: *Comparison Template Functions*

```
// TEMPLATE FUNCTION lexicographical_compare
template<class InIt1, class InIt2> inline
    bool lexicographical_compare(InIt1 first1, InIt1 last1,
        InIt2 first2, InIt2 last2)
    {for (; first1 != last1 && first2 != last2;
        ++first1, ++first2)
        if (*first1 < *first2)
            return (true);
        else if (*first2 < *first1)
            return (false);
    return (first1 == last1 && first2 != last2); }

// TEMPLATE FUNCTION mismatch
template<class InIt1, class InIt2> inline
    pair<InIt1, InIt2> mismatch(InIt1 first, InIt1 last, InIt2 x)
    {for (; first != last && *first == *x; ++first, ++x)
        ;
    return (pair<InIt1, InIt2>(first, x)); }

// TEMPLATE FUNCTION equal
template<class InIt1, class InIt2> inline
    bool equal(InIt1 first, InIt1 last, InIt2 x)
    {return (mismatch(first, last, x).first == last); }
//End of File
```


Listing 4: *Searching Template Functions*

```
// TEMPLATE FUNCTION find
template<class InIt, class T> inline
    InIt find(InIt first, InIt last, const T& val)
    {for (; first != last; ++first)
        if (*first == val)
            break;
    return (first); }

// TEMPLATE FUNCTION adjacent_find
template<class FwdIt> inline
    FwdIt adjacent_find(FwdIt first, FwdIt last)
    {for (FwdIt firstb; (firstb = first) != last
        && ++first != last; )
        if (*firstb == *first)
            return (firstb);
    return (last); }

// TEMPLATE FUNCTION count
template<class InIt, class T> inline
/// iterator_traits<InIt>::distance_type
///        count(InIt first, InIt last, const T& val)
///    {iterator_traits<InIt>::distance_type n = 0;
    size_t count(InIt first, InIt last, const T& val) ///
    {size_t n = 0;    ///
    for (; first != last; ++first)
        if (*first == val)
            ++n;
    return (n); }

// TEMPLATE FUNCTION search
template<class FwdIt1, class FwdIt2> inline
    FwdIt1 search(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2)
    {return (_Search(first1, last1, first2, last2,
        _Dist_type(first1), _Dist_type(first2))); }
template<class FwdIt1, class FwdIt2,
    class Diff1, class Diff2> inline
    FwdIt1 _Search(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2, Diff1 *, Diff2 *)
    {Diff1 d1 = 0;
    _Distance(first1, last1, d1);
    Diff2 d2 = 0;
    _Distance(first2, last2, d2);
    for (; d2 <= d1; ++first1, --d1)
        {FwdIt1 x1 = first1;
        for (FwdIt2 x2 = first2; ; ++x1, ++x2)
            if (x2 == last2)
                return (first1);
            else if (!(*x1 == *x2))
                break; }
    return (last1); }

// TEMPLATE FUNCTION search_n
template<class FwdIt1, class Diff2, class T> inline
    FwdIt1 search_n(FwdIt1 first1, FwdIt1 last1, Diff2 n,
        const T& val)
    {return (_Search_n(first1, last1, n,
        val, _Dist_type(first1))); }
template<class FwdIt1, class Diff2, class T, class Diff1> inline
    FwdIt1 _Search_n(FwdIt1 first1, FwdIt1 last1,
        Diff2 n, const T& val, Diff1 *)
    {Diff1 d1 = 0;
    _Distance(first1, last1, d1);
    for (; n <= d1; ++first1, --d1)
        {FwdIt1 x1 = first1;
        for (Diff2 d2 = n; ; ++x1, --d2)
            if (d2 == 0)
                return (first1);
            else if (!(*x1 == val))
                break; }
    return (last1); }

// TEMPLATE FUNCTION find_end
template<class FwdIt1, class FwdIt2> inline
    FwdIt1 find_end(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2)
    {return (_Find_end(first1, last1, first2, last2,
        _Dist_type(first1), _Dist_type(first2))); }
template<class FwdIt1, class FwdIt2,
    class Diff1, class Diff2> inline
    FwdIt1 _Find_end(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2, Diff1 *, Diff2 *)
```

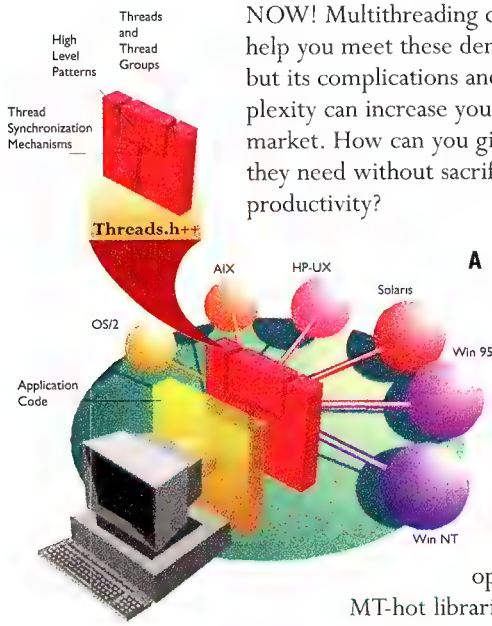
INTRODUCING

Version 1.0

A Simpler Way to Write Portable Multithreaded Apps

Threads.h++

The C++ class library for writing portable MT-safe and MT-hot libraries and applications



Today's sophisticated end users demand more from their applications–power, throughput, responsiveness. And they want it NOW! Multithreading can help you meet these demands, but its complications and complexity can increase your time to market. How can you give them what they need without sacrificing your own productivity?

A Win/Win Situation

Everybody wins with Threads.h++, the cross-platform solution! Threads.h++ provides portable object-oriented mechanisms to simplify the development of MT-safe and MT-hot libraries and applications. Use the library's encapsulation mechanisms to increase your productivity, while supercharging your apps with improved throughput and responsiveness.

Write Once, Port Often

With the Threads.h++ platform-independent, generic multithreading model, write your multithreaded app once, then recompile to run in other environments. Win NT, Win 95, Solaris, HP-UX, AIX, OS/2: the choice is yours!

Synchronize Your Threads

You can make your MT-hot app safe when you coordinate interthread synchronization and communication with Threads.h++. The library's complete set of thread synchronization mechanisms includes critical sections; simple, recursive, and FIFO mutexes; semaphores; readers-writer locks; condition variables; and barriers.

Fine-tune Your Application

Use the library's thread creation and management classes to take advantage of platform-specific threading features. Check the status of threads, create and destroy threads, and query or adjust the scheduling state of threads to fine-tune your application. Call now to find out how Threads.h++ can increase your productivity.

The Software Parts Company™

Rogue Wave SOFTWARE

Surf Us http://www.roguewave.com

Call Us (800) 487-3217 (541) 754-5010

Fax Us (541) 757-6650

Email Us threads@roguewave.com

Sequence Editing

There are various ways to edit a sequence. For the following functions, the code is sufficiently obvious that I won't show it here:

- `replace` replaces each element with the value you specify with another value you specify.
- `replace_if` replaces each element that satisfies the predicate you specify with a value you specify.

Listing 4: *continued*

```
    {Diff1 d1 = 0;
    _Distance(first1, last1, d1);
    Diff2 d2 = 0;
    _Distance(first2, last2, d2);
    FwdIt1 ans = last1;
    if (0 < d2)
        for (; d2 <= d1; ++first1, --d1)
            {FwdIt1 x1 = first1;
            for (FwdIt2 x2 = first2; ; ++x1)
                if (!(*x1 == *x2))
                    break;
                else if (++x2 == last2)
                    {ans = first1;
                    break; }}
    return (ans); }

// TEMPLATE FUNCTION find_first_of
template<class FwdIt1, class FwdIt2> inline
    FwdIt1 find_first_of(FwdIt1 first1, FwdIt1 last1,
        FwdIt2 first2, FwdIt2 last2)
    {for (; first1 != last1; ++first1)
        for (FwdIt2 x2 = first2; x2 != last2; ++x2)
            if (*first1 == *x2)
                return (first1);
    return (first1); }
//End of File
```


Listing 5: *Transforming Template Functions*

```
// TEMPLATE FUNCTION for_each
template<class InIt, class Fun> inline
    Fun for_each(InIt first, InIt last, Fun op)
    {for (; first != last; ++first)
        op(*first);
    return (op); }

// TEMPLATE FUNCTION generate
template<class FwdIt, class Fun> inline
    void generateFFwdIt first, FwdIt last, Fun fun)
    {for (; first != last; ++first)
        *first = fun(); }

// TEMPLATE FUNCTION generate_n
template<class OutIt, class Diff, class Fun> inline
    void generate_n(OutIt first, Diff n, Fun fun)
    {for (; 0 < n; --n, ++first)
        *first = fun(); }

// TEMPLATE FUNCTION transform WITH UNARY OP
template<class InIt, class OutIt, class Unop> inline
    OutIt transform(InIt first, InIt last, OutIt x, Unop unop)
    {for (; first != last; ++first, ++x)
        *x = unop(*first);
    return (x); }

// TEMPLATE FUNCTION transform WITH BINARY OP
template<class InIt1, class InIt2, class OutIt, class Bop> inline
    OutIt transform(InIt1 first1, InIt1 last1, InIt2 first2,
        OutIt x, Bop op)
    {for (; first1 != last1; ++first1, ++first2, ++x)
        *x = op(*first1, *first2);
    return (x); }
//End of File
```

Listing 6: *Unique Template Functions*

```
// TEMPLATE FUNCTION unique_copy
template<class InIt, class OutIt> inline
    OutIt unique_copy(InIt first, InIt last, OutIt x)
    {return (first == last ? x :
        _Unique_copy(first, last, x,
            _Iter_cat(first))); }
template<class InIt, class OutIt> inline
    OutIt _Unique_copy(InIt first, InIt last, OutIt x,
        input_iterator_tag)
    {return (_Unique_copy(first, last, x, _Val_type(first))); }
template<class InIt, class OutIt, class T> inline
    OutIt _Unique_copy(InIt first, InIt last, OutIt x, T *)
    {T val = *first;
    for (*x++ = val; ++first != last; )
        if (!(val == *first))
            val = *first, *x++ = val;
    return (x); }
template<class FwdIt, class OutIt> inline
    OutIt _Unique_copy(FwdIt first, FwdIt last, OutIt x,
        forward_iterator_tag)
    {FwdIt firstb = first;
    for (*x++ = *firstb; ++first != last; )
        if (!(*firstb == *first))
            firstb = first, *x++ = *firstb;
    return (x); }
template<class BidIt, class OutIt> inline    ///
    OutIt _Unique_copy(BidIt first, BidIt last, OutIt x,    ///
        bidirectional_iterator_tag)    ///
    {return (_Unique_copy(first, last, x,    ///
        forward_iterator_tag())); }    ///
template<class RanIt, class OutIt> inline    ///
    OutIt _Unique_copy(RanIt first, RanIt last, OutIt x,    ///
        random_access_iterator_tag)    ///
    {return (_Unique_copy(first, last, x,    ///
        forward_iterator_tag())); }    ///

// TEMPLATE FUNCTION unique
template<class FwdIt> inline
    FwdIt unique(FwdIt first, FwdIt last)
    {first = adjacent_find(first, last);
    return (unique_copy(first, last, first)); }
//End of File
```

Watcom C/C++. Performance on any Platform.

Watcom C/C++ accelerates development of lightning-fast, multi-platform 16 and 32 bit applications. Reliable, high-performance code generation and consistent C and C++ language implementation are delivered across all supported platforms, making it easy to develop applications for several targets from a common source code base. In a single package, Watcom C/C++ provides a comprehensive development environment with the tools, SDKs and libraries you need to create powerful applications for popular PC platforms including Windows®, OS/2® Warp, Windows® 95, and Windows NT™.

To find out more about Watcom C/C++, call us at 1-800-265-4555 or visit ***http://www.powersoft.com***

Powersoft®

Watcom C/C++ v10.6 Features

- Target Platforms: Windows 95, Windows NT, Windows 3.x, Win32s, OS/2 Warp, OS/2 2.x, Extended DOS, Novell NLM, OS/2 1.x, DOS
- 16 and 32 bit C and C++ compilers with industry-leading optimization technology
- Comprehensive MFC support with source, samples, and documentation
- Visual Programmer by Blue Sky Software for rapid MFC application development
- Multi-platform toolset with a consistent interface across multiple platforms

- `replace_copy` behaves like `replace`, except that you also specify where to store the modified sequence.
- `replace_copy_if` behaves like `replace_if`, except that you also specify where to store the modified sequence.
- `remove` removes each element with the value you specify, by copying down subsequent elements.
- `remove_if` removes each element that satisfies the predicate you specify, by copying down subsequent elements.
- `remove_copy` removes each element with the value you specify, by omitting such elements as it copies a sequence.

Listing 7: *Rearranging Template Functions*

```
// TEMPLATE FUNCTION reverse
template<class BidIt> inline
    void reverse(BidIt first, BidIt last)
    {_Reverse(first, last, _Iter_cat(first)); }
template<class BidIt> inline
    void _Reverse(BidIt first, BidIt last,
        bidirectional_iterator_tag)
    {for (; first != last && first != --last; ++first)
        iter_swap(first, last); }
template<class RanIt> inline
    void _Reverse(RanIt first, RanIt last,
        random_access_iterator_tag)
    {for (; first < last; ++first)
        iter_swap(first, --last); }

// TEMPLATE FUNCTION reverse_copy
template<class BidIt, class OutIt> inline
    OutIt reverse_copy(BidIt first, BidIt last, OutIt x)
    {for (; first != last; ++x)
        *x = *--last;
    return (x); }

// TEMPLATE FUNCTION rotate
template<class FwdIt> inline
    void rotate(FwdIt first, FwdIt mid, FwdIt last)
    {if (first != mid && mid != last)
        _Rotate(first, mid, last, _Iter_cat(first)); }
template<class FwdIt> inline
    void _Rotate(FwdIt first, FwdIt mid, FwdIt last,
        forward_iterator_tag)
    {for (FwdIt x = mid; ; )
        {iter_swap(first, x);
        if (++first == mid)
            if (++x == last)
                break;
            else
                mid = x;
        else if (++x == last)
            x = mid; }}
template<class BidIt> inline
    void _Rotate(BidIt first, BidIt mid, BidIt last,
        bidirectional_iterator_tag)
    {reverse(first, mid);
    reverse(mid, last);
    reverse(first, last); }
template<class RanIt> inline
    void _Rotate(RanIt first, RanIt mid, RanIt last,
        random_access_iterator_tag)
    {_Rotate(first, mid, last, _Dist_type(first),
        _Val_type(first)); }
template<class RanIt, class Diff, class T> inline
    void _Rotate(RanIt first, RanIt mid, RanIt last, Diff *, T *)
    {Diff d = mid - first;
    Diff n = last - first;
    for (Diff i = d; i != 0; )
        {Diff j = n % i;
        n = i, i = j; }
    if (n < last - first)
        for (; 0 < n; --n)
            {RanIt x = first + n;
            RanIt y = x;
            T val = *x;
            RanIt z = y + d == last ? first : y + d;
            while (z != x)
                {*y = *z;
                y = z;
                z = d < last - z ? z + d
                    : first + (d - (last - z)); }
            *y = val; }}

// TEMPLATE FUNCTION rotate_copy
template<class FwdIt, class OutIt> inline
    OutIt rotate_copy(FwdIt first, FwdIt mid, FwdIt last, OutIt x)
    {x = copy(mid, last, x);
    return (copy(first, mid, x)); }

// TEMPLATE FUNCTION random_shuffle
template<class RanIt> inline
    void random_shuffle(RanIt first, RanIt last)
    {if (first != last)
        _Random_shuffle(first, last, _Dist_type(first)); }
template<class RanIt, class Diff> inline
    void _Random_shuffle(RanIt first, RanIt last, Diff *)
    {const int _RBITS = 15;
    const int _RMAX = (1U << _RBITS) - 1;
    RanIt x = first;
    for (Diff d = 1; ++x != last; ++d)
        {unsigned long ranm = _RMAX;
        unsigned long rann = rand() & _RMAX;
        for (; ranm < d && ranm != ~0UL;
            ranm = ranm << _RBITS | _RMAX)
            rann = rann << _RBITS | _RMAX;
        iter_swap(x, first + Diff(rann % d)); }}
template<class RanIt, class Ranf> inline
    void random_shuffle(RanIt first, RanIt last, Ranf& ranf)
    {if (first != last)
        _Random_shuffle(first, last, ranf, _Dist_type(first)); }
template<class RanIt, class Ranf, class Diff> inline
    void _Random_shuffle(RanIt first, RanIt last,
        Ranf& ranf, Diff *)
    {RanIt x = first;
    for (Diff d = 1; ++x != last; ++d)
        iter_swap(x, first + Diff(ranf(d))); }
//End of File
```

- `remove_copy_if` removes each element that satisfies the predicate you specify, by omitting such elements as it copies a sequence.

Listing 6 shows two somewhat more interesting functions that edit a sequence. `unique_copy` omits all but the first element in any subsequence of elements that compare equal, as it copies the sequence. Unix fans will recognize this as the heart of the classic software tool `uniq`. The code presented here illustrates the use of `_Iter_cat` to select different approaches based on the category of iterators supplied as function arguments.

Working with input iterators, the function must store an actual copy of an element value. It cannot refer to an earlier value through an iterator once that value has passed by. But working with forward (or stronger) iterators, the function can simply store a copy of the iterator to refer to a value earlier in the sequence. There is less risk that the function will be asked to construct, store, and destroy an object that is expensive in storage or execution time.

In principle, there is no need for the last two overloads. I have found, however, that not all compilers know to select the forward-iterator version for bidirectional or random-access iterators. Thus, the redundant versions are "commented in" as a practical necessity for the nonce.

Template function `unique` performs in place much the same function as `unique_copy`. In this case, the latter function can do all the heavy lifting for the former, once `unique` determines that any rearranging is necessary. It calls `adjacent_find` for a relatively quick answer to that question.

As usual, both `unique_copy` and `unique` have versions that accept an added predicate argument as well.

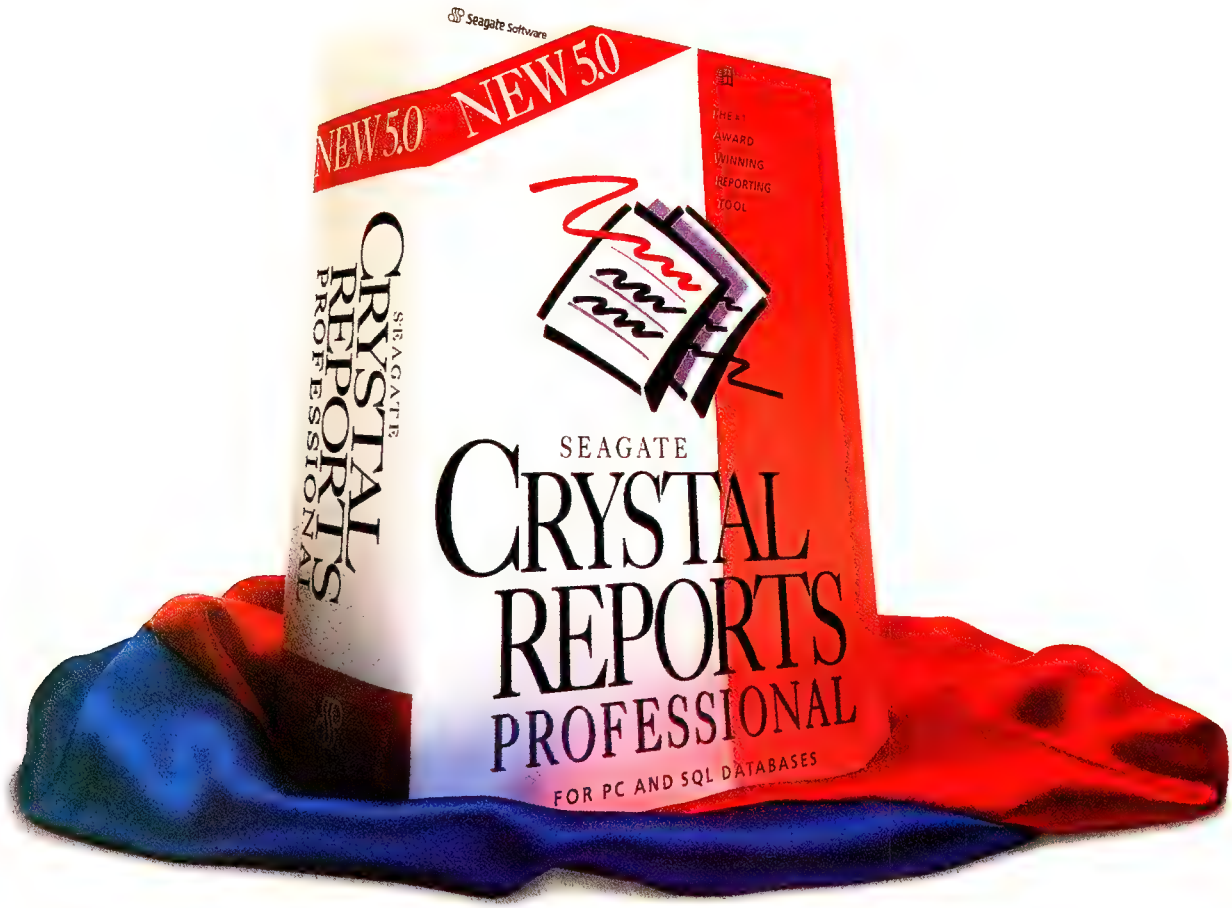
Rearranging Elements

The last group of algorithms I show this month are lumped together in Listing 7. All of these rearrange the elements of a sequence in various useful ways. `reverse` and `reverse_copy` are the simplest. They reverse the elements, either in place or while copying. The former takes advantage of a slight optimization when handed random-access iterators instead of bidirectional ones.

Rotating a sequence is rather harder, at least if you do the job in place. I believe that template function `rotate` holds the record for taking most ambitious advantage of the ability to fan out on iterator category. I encourage you to study all three versions of this function. They are quite distinct approaches, and each is clever in its own right.

By comparison, `rotate_copy` is rather tame. In the coming months, you will see how other algorithms take advantage of the greater simplicity of rotating while copying, when the opportunity presents itself to do so.

Finally, `random_shuffle` does as its name implies. It rearranges the elements of a sequence by performing pairwise swaps of each element with another element chosen randomly. Most of the complexity here stems from a half-hearted attempt to make the algorithm more general. The code endeavors to use the C library function `rand` to generate a random number big enough to deal with an arbitrary iterator distance type. All it succeeds in doing, however, is extend a 15-bit function to generate a random `unsigned long` value. Probably that's good enough for most applications. I haven't yet tried to solve the fully general problem. ❑



THE LAST TIME A REPORTER MADE THIS KIND OF TRANSFORMATION, HE WORE TIGHTS AND A BIG "S" ON HIS CHEST.

Get ready for a giant leap in reporting capabilities. Because now there's new Crystal Reports Professional 5.0 — giving you the power to develop and report like never before.

This major release offers an incredible new level of reporting flexibility. With Crystal Reports 5.0, you can create anything from subreports to form-style reports, conditional reports, multiple detail section reports, Web-ready reports, multiple summary cross-tab reports, BackOffice® reports and more.

You also have the added flexibility to publish presentation quality reports directly to the Internet and Intranets using new HTML output options. Plus, you get additional power and control with over 25 new functions, over 15 new ActiveX® (OLE/OCX) properties and many other super new features.

To see Crystal Reports 5.0, visit our Web site: //www.img.seagatesoftware.com/. Then give us a call. You may qualify for the special upgrade price of just $199 — that's almost $200 off the regular retail price. And, you'll receive our 60-day money-back guarantee, which makes your decision to upgrade all the more bullet-proof.

Order your transformed reporter today. **Call 1-800-877-2340 dept. CR18.**

Seagate Software

Information Management Group

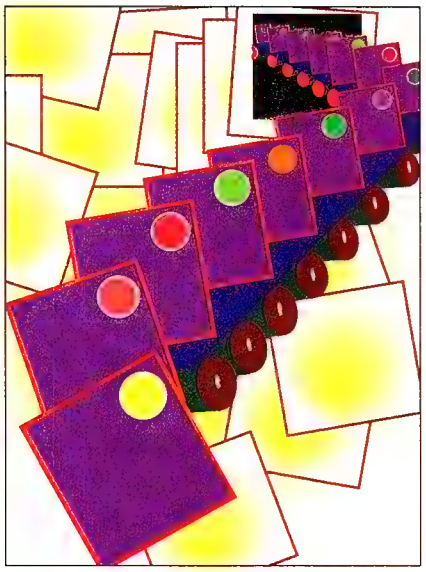
Formerly CRYSTAL *A Seagate Software Company*

Other inquiries call 604-681-3435. Email-sales@crystalinc.com. Facts-on-demand 604-681-3450. Now available: Crystal Reports 5.0 Interactive Learning CD from Compunnel Software Group Inc. Crystal Reports is a trademark of Seagate Software Inc. BackOffice and ActiveX are trademarks of Microsoft Corporation. Superman and all related characters, names and indicias are trademarks of DC Comics.

Linda Kasparek

Processing Variant Records with STL

When you need a map class, you need a map class, and STL provides a good one for many applications.



Introduction

Processing variant records, which is often carried out via the brute force approach (a la the big `switch` statement), can lead to cumbersome code. This code becomes difficult to develop and maintain when the `switch` statement gets very large. Also, since the `switch` statement is essentially a "hard-wired" reflection of the record formats to be processed, it provides little flexibility for modifications or additions. Fortunately, C++ gives us a way to handle variant records a bit more gracefully. By storing information about the different record formats and their fields in an STL `map`, the mechanics can be reduced to a few lines of code. In this article, I show how to use STL to tame the unwieldy `switch` statement, and how to concentrate data format definitions in one convenient location within the source code.

The Problem with Variant Records

A variant record, as represented in C/C++, is a `struct` with a union of the variants. The fixed part of the record usually contains a field that specifies the format of the variant part.

A typical variant record structure would look something like this:

```
typedef struct {
    char desc[31];
    char fmt;    // 'A', 'B', or 'C'
    union {
        struct {
            // pseudo code
            field1
            field2
            field3
        // to be used if fmt = 'A'
        } A;
        struct {
            // pseudo code
            field1
            field2
            field3
            field4
        // to be used if fmt = 'B'
        } B;
        struct {
            // pseudo code
            field1
            field2
        // to be used if fmt = 'C'
        } C;
    } Variant;
} Record;
```

Note that `field1` in `A`, `B`, and `C` may not necessarily be the same field.

Variant records in C/C++ are often used to store information from the outside world — commonly a database or configuration file. As an example, I show a simple data file with three different record formats (see Figure 1). In this file, a record spans multiple lines, with one line per field; each record is delimited by the '^' character. Each field is prepended by a field identifier. The file uses these identifiers to allow fields in the variant part to occur in any order, or to be omitted if the variant part is a default value. The first two fields are fixed:

```
N - name
F - record format
```

The 'F' prefix in front of the second field indicates that this is a format ID, which identifies the variant portion of the record. Table 1 shows the various field identifiers used within each format.

The structure that will hold these records is shown in Listing 1, as is the brute force approach to reading these records into the structure.

As you can see in Listing 1, the brute force approach requires a pretty ugly switch statement — the code has to branch on each record format and then branch on each field within the record format. This approach would become too unwieldy to maintain if it got any more complicated — and in real life it almost always does! Every time a record format changed, you'd have to muck around with this section of code and you'd probably want to regression test the parts that didn't change. It's easy to introduce new bugs with forgotten `break` statements, for example. So, you say, there's got to be a better way!

A Solution

There is a more elegant way to handle variant records, using STL's `map` container class and the C/C++ macro `offsetof`. The `map` container provides storage and retrieval of objects via keys. It holds data in the form of Key/Value pairs, where the Value is some object to be stored in the map, and Key is another object used to identify a particular Value object. `offsetof` returns the offset in bytes of a member relative to the base address of its parent structure. The basic idea here is to store the offset as the

Linda Kaparek has what she considers the ultimate job — developing a Windows-based mass-market software package for a large midwestern financial corporation while working from home. She is pursuing an M.S. in Computer Science at UMass-Lowell, having obtained an M.S. in Statistics in 1993. She has been in software development for 11 years, the most recent three years in MS Windows/C++.

Value and to combine the record format and field prefix to make the Key. When the program hands a Key to the map, the map hands back an offset. The offset will be used to point to the corresponding structure member into which the data will be copied. This scheme is illustrated in Figure 2.

Figure 1: *Data file containing variant records*

```
NC/C++
F1
$100.99
D04/03/96
^
NUsers
F2
D05/17/96
TLa de da!
^
NJournal
F3
$800.25
D12/25/96
THum de dum!
```

Table 1: *Field indentifiers*

Record Format 1:
$ – amount
T – description
Record Format 2:
$ – amount
D – date
Record Format 3:
$ – amount
D – date
T – description

Listing 1: *Variant record* `struct` *and* `switch` *statement used to populate it*

```
typedef struct {
    char Name[21];
    short Format;
    union {
        struct {
            float Amount;
            char Date[9];
        } sFormat1;
        struct {
            char Date[9];
            char Desc[31];
        } sFormat2;
        struct {
            float Amount;
            char Date[9];
            char Desc[30];
        } sFormat3;
    }
} sVariantRecord;

// switch statement used to process
// above record

switch (recordFormat) {
case FORMAT1:
        switch (prefix) {
        case '$':
            record.sFormat1.Amount =
                atof(p);
            break;
        case 'D':
            strcpy(record.sFormat1.Date, p);
            break;
        }
        break;
    case FORMAT2:
        switch (prefix) {
        case 'D':
            strcpy(record.sFormat2.Date, p);
            break;
        case 'T':
            strcpy(record.sFormat2.Desc, p);
            break;
        }
        break;
    case FORMAT3:
        switch (prefix) {
        case '$':
            record.sFormat3.Amount =
                atof(p);
            break;
        case 'D':
            strcpy(record.sFormat3.Date, p);
            break;
        case 'T':
            strcpy(record.sFormat3.Desc, p);
            break;
        }
        break;
    }

// End of File
```

To instantiate a map container class you need to define a pair of classes — a class to represent keys and a class to represent their corresponding values. You also need to provide for a comparison function object. STL uses the comparison function object, specified by providing a comparison class as the third parameter in the template instantiation, for `find` and `insert` operations. The predefined comparison class `less<T>`, found in STL's `function.h`, uses `operator<` to make comparisons. For keys of a built-in type, you get this for free. For user-defined types, there are two alternatives — either write a custom comparison function class, or override `operator<` in the key and use `less<T>` where `T` refers to the key class. In this article, I show how to devise a comparison function class.

Implementing the Variant Map

First, I'll implement the key class. This class is pretty straightforward; it has two attributes, the record format and the field prefix. I also define a copy and default constructor, an assignment operator, a default destructor, and a constructor that takes arguments used to initialize the two attributes. The STL `map` class requires that the first four functions be explicitly defined. The code for this class (`CKey`) is

in the `globals.h` header file, shown in Listing 2.

The comparison function class is also simple if you examine the key's attributes separately. The code for this class is also in `globals.h`, just below the implementation of the `CKey` class. This class has just one member: `operator()`. Defining `operator()` enables STL to use this class as a function object. This implementation of `operator()` works as follows: first, it compares the record format attribute for each key and returns `TRUE` or `FALSE`, depending on which is the greater. If the values for the record formats are equal, then `operator()` examines the prefix and returns `TRUE` or `FALSE` depending on which is the greater.

The value class `CValue` is straightforward as well. It has two attributes, the offset and the type indicator. The type indicator will come in handy when it is time to copy the value into the pointer location determined by the offset. If the value is a string, the program uses `strcpy`. Otherwise, it converts the value into a number using `atoi`, `atof`, or some other appropriate conversion and then copies it to the target field. This class also meets the prerequisite type requirements by defining the default and copy constructors, default destructor, and assignment operator.

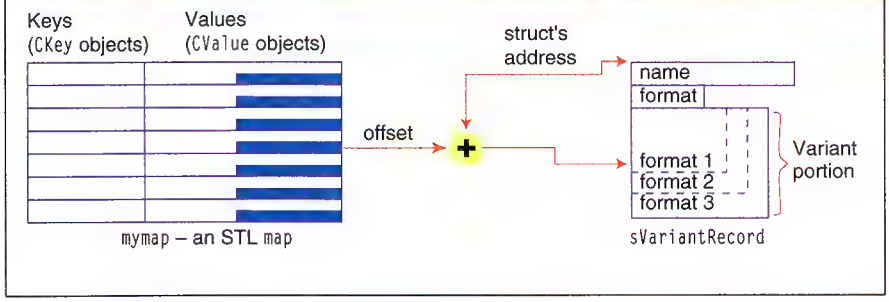
To populate the map, I declare an array of `CKey/CValue` pair objects and then instantiate the map like this:

```
mymap m(array, array + 7);
```

`mymap` is a typedef for `map<CKey, CValue, less_CKey>`. The above statement calls a map constructor that takes two pointers to designate an initialization range. This invokes a map initializer that inserts each pair element in the array (`array`) into the map. Another alternative would be to declare and define each pair and insert it into the map one at a time.

I created the `File` class to encapsulate the reading and managing of variant record files; see `file.h` and `file.cpp` in Listings 3 and 4, respectively. `File` is responsible for getting each complete record from the file. To do this, `File` has to read each line in the record, look up the offset from the map based on the record format and field prefix, copy the value into the corresponding structure member to which the offset points, and return the

Figure 2 *Accessing a variant record via a stored offset*



populated structure. To look up the offset in the map, File uses the iterator j, declared in mapdef.cpp (Listing 5), and calls the map's find() function with the key j = m.find(key);. find returns the Key/Value pair object. The pair_type template class is a typedef for pair<CKey, CValue>, which defines the members first and second, that contain the key and value objects, respectively.

To see all this in action, take a look at the little driver program (Listing 6) I implemented in Visual C++ 4.1. It instantiates a File

Listing 2: *Defines key and value classes and variant record* struct

```
#ifndef _GLOBALS_H
#define _GLOBALS_H

#ifdef _USEMAP
#include <map.h>
#endif

typedef unsigned int BOOL;
const int FALSE = 0;
const int TRUE = 1;

// record formats
const unsigned int FORMAT1 = 1;
const unsigned int FORMAT2 = 2;
const unsigned int FORMAT3 = 3;

#ifdef _USEMAP
// datatypes
const unsigned int FLOAT = 1;
const unsigned int STRING = 2;

class CKey {
public:
    CKey() {}
    CKey(unsigned int Format,
         char Prefix) : m_format(Format), m_prefix(Prefix) {}
    ~CKey() {}

    int operator=(const CKey& rhs)
        { m_format = rhs.m_format; m_prefix = rhs.m_prefix; return 1; }
    CKey(const CKey& rhs)
        { m_format = rhs.m_format; m_prefix = rhs.m_prefix; }

    unsigned int m_format;
    char m_prefix;
};

class less_CKey {
public:
    BOOL operator()(const CKey& K1, const CKey& K2) const
    {
        if (K1.m_format > K2.m_format)
            return TRUE;
        else
        if (K1.m_format < K2.m_format)
            return FALSE;
        else
        if (K1.m_prefix > K2.m_prefix)
            return TRUE;
        else
            return FALSE;
    }
};

class CValue {
public:
    CValue() {}
    CValue(long Offset, unsigned int Datatype) : m_offset(Offset),
         m_datatype(Datatype) {}
    ~CValue() {}

    int operator=(const CValue& rhs)
        { m_offset = rhs.m_offset; m_datatype = rhs.m_datatype; return 1; }
    CValue(const CValue& rhs)
        { m_offset = rhs.m_offset; m_datatype = rhs.m_datatype; }

    unsigned int Datatype() { return m_datatype; }

    long m_offset;
    unsigned int m_datatype;
};

typedef map < CKey, CValue, less_CKey > mymap;

typedef pair< CKey, CValue > pair_type;
extern mymap m;
extern mymap::iterator j;
#endif    // #ifdef _USEMAP

const int cDateLen = 8;
const int cDescLen = 30;

typedef struct {
    char Name[21];
    short Format;
    union {
        struct {
            float Amount;
            char Date[cDateLen + 1];
        } sFormat1;
        struct {
            char Date[cDateLen + 1];
            char Desc[cDescLen + 1];
        } sFormat2;
        struct {
            float Amount;
            char Date[cDateLen + 1];
            char Desc[cDescLen + 1];
        } sFormat3;
    };
} sVariantRecord;


#endif    // _GLOBALS_H
// End of File
```

object and calls its GetRecord member function until all records have been read. The driver prints out each populated record structure that GetRecord returns. The output appears at the bottom of Listing 6.

Other Uses for this Technique

I've had other occasions to use this scheme — if only I'd been able to get my hands on STL. For instance, I had a task to populate a flat structure with inputs from a front-end that could only pass it one field at a time — it did not have the capability to pass an entire populated structure. I then used this flat structure to instantiate objects in the back-end for processing. At the time, STL was not available. The field IDs were constant, consecutive integers, so I created an array of structures and could access the array using the field ID as the index. (We avoided using compiler-specific map classes, to maintain portability.) Anyway, having the STL support would have been nice.

Listing 3: *The File class*

```
#ifndef _FILE_H
#define _FILE_H

#include <fstream.h>
#include "Globals.h"

class File
{
public:
    File();
    ~File();

    void Init(char* FileName);
    BOOL GetRecord(sVariantRecord& record);

private:
    ifstream* m_file;
};
#endif
// End of File
```

Listing 4: *Implements file processing*

```
#include <stdlib.h>
#include <string.h>
#include "Globals.h"
#include "File.h"

File::File()
: m_file(0)
{
}

File::~File()
{
    if (m_file) {
        m_file->close();
        delete m_file;
    }
}

void File::Init(char* FileName)
{
    m_file = new ifstream;
    m_file->open(FileName, ios::in);
//    int num = m.size();
}

BOOL File::GetRecord(sVariantRecord& record)
{
    char prefix, lpszTokenLine[81];
    short recordFormat;
    float number;
    char* here, *p;
    short datatype;
    CKey key;

    // priming read
    m_file->getline(lpszTokenLine, 81);
    if (m_file->eof()) {
        return FALSE;
    }
    prefix = lpszTokenLine[0];
    p = &lpszTokenLine[1];

    while (prefix != '^') {
        switch (prefix) {
        case 'N':
            strcpy(record.Name, p);
            break;
        case 'F':
            recordFormat = atoi(p);
            record.Format = recordFormat;
            break;
        default:
            key.m_format = recordFormat;
            key.m_prefix = prefix;
            j = m.find(key);
            if (j != m.end()) {
                datatype =
                    (*j).second.m_datatype;
```

- C++ class library
- No HTML experience required
- Supports common HTML 3.0 features
- Supports full HTML 2.0 Spec
- ANSI/ISO compatible
- Easy to Use
- Helps eliminate common mistakes
- Shortens development time
- Operates with all popular versions of ANSI/ISO Standard Template Library and String

ObjectSpace

CALL 1.800.OBJECT.1

email: sales@objectspace.com

Request Reader Service #122

Conclusion

As you may have noticed, using STL didn't get me completely away from the dreaded `switch` statement. Function `File::GetRecord` (Listing 4) still needs to do some processing on format types to populate a variant record. The difference between this `switch` statement and the "brute force" `switch` is subtle, but important. The brute force `switch` requires a separate case clause for every format type, and a separate case under each format for every field type. The improved `switch` requires a separate case clause only for each field type. This technique basically splits the handling of formats and the handling of field types into two separate, manageable chunks. The format types are more or less

Listing 4: *continued*

```
                switch (datatype) {
                case FLOAT:
                    here = (char*)&record + (*j).second.m_offset;
                    number = atof(p);
                    memcpy(here, &number, sizeof(float));
                    break;
                case STRING:
                    here = (char*)&record + (*j).second.m_offset;
                    strcpy(here, p);
                    break;
                default:
                    // some error
                    break;
                }    // switch
            }
            break;
        }    // switch
        m_file->getline(lpszTokenLine, 81);
        prefix = lpszTokenLine[0];
        p = &lpszTokenLine[1];
    }
    return TRUE;
}

//omitted: brute force variant population method.  Full source code
//is available on code disk and via ftp, see page 3 for details --mb
//End of File
```


Listing 5: *Creation of Key/Value pairs*

```
#ifdef _USEMAP
#include "Globals.h"

CKey key1(FORMAT1, '$');
CValue value1(offsetof(sVariantRecord, sFormat1.Amount), FLOAT);
pair_type p1 (key1, value1);

CKey key2(FORMAT1, 'D');
CValue value2(offsetof(sVariantRecord, sFormat1.Date), STRING);
pair_type p2 (key2, value2);

CKey key3(FORMAT2, 'D');
CValue value3(offsetof(sVariantRecord, sFormat2.Date), STRING);
pair_type p3 (key3, value3);

CKey key4(FORMAT2, 'T');
CValue value4(offsetof(sVariantRecord, sFormat2.Desc), STRING);
pair_type p4 (key4, value4);

CKey key5(FORMAT3, '$');
CValue value5(offsetof(sVariantRecord, sFormat3.Amount), FLOAT);
pair_type p5 (key5, value5);

CKey key6(FORMAT3, 'D');
CValue value6(offsetof(sVariantRecord, sFormat3.Date), STRING);
pair_type p6 (key6, value6);

CKey key7(FORMAT3, 'T');
CValue value7(offsetof(sVariantRecord, sFormat3.Desc), STRING);
pair_type p7 (key7, value7);

pair_type array [] =
{
    p1,
    p2,
    p3,
    p4,
    p5,
    p6,
    p7,
};

mymap m(array, array + 7);
mymap::iterator j;
#endif//End of File
```

Listing 6: *Test driver program and output*

```
#include <fstream.h>
#include <memory.h>
#include "Globals.h"
#include "file.h"

void main()
{
    File file;
    file.Init("data.dat");
    BOOL bStatus;
    fstream out;
    out.open("out.txt", ios::out);
    cout = out;

    sVariantRecord rec;
    memset(&rec, 0, sizeof(sVariantRecord));

    bStatus = file.GetRecord(rec);
    while (bStatus != NULL) {
        cout << "Name: " << rec.Name << '\n';
```

automatically handled by modifying the `struct sVariantRecord`, and perhaps adding a few lines to `mapdef.cpp`. (Adding a format requires adding four lines to `mapdef.cpp`; modifying a format requires no changes to this file.) Field types, which are far less likely to change over the life of an application, are handled in the `switch` statement.

I suppose I could have pulled this off in C using a couple of associative arrays. But STL let me program intuitively, by defining Key/Value pair objects and comparison objects. It was just too portable and elegant a solution to resist. ❑

Reference

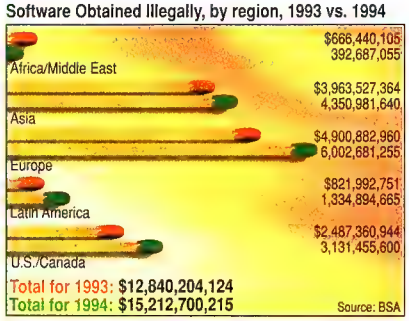
Mark Nelson, *C++ Programmer's Guide to the Standard Template Library*. (IDG Books Worldwide, Inc., 1995).

Listing 6: *continued*

```
cout << "Format: " << rec.Format << '\n';
switch (rec.Format) {
case FORMAT1:
    cout << "Amount: " << rec.sFormat1.Amount << '\n';
    cout << "Date: " << rec.sFormat1.Date << '\n';
    break;
case FORMAT2:
    cout << "Date: " << rec.sFormat2.Date << '\n';
    cout << "Desc: " << rec.sFormat2.Desc << '\n';
    break;
case FORMAT3:
    cout << "Amount: " << rec.sFormat3.Amount << '\n';
    cout << "Date: " << rec.sFormat3.Date << '\n';
    cout << "Desc: " << rec.sFormat3.Desc << '\n';
    break;
default:
    cout << "ho hum" << '\n';
    break;
}
cout << '\n';
memset(&rec, 0, sizeof(sVariantRecord));
bStatus = file.GetRecord(rec);
    }
    out.close();
}

//output:

Name: C/C++
Format: 1
Amount: 100.99
Date: 04/03/96

Name: Users
Format: 2
Date: 05/17/96
Desc: La de da!

Name: Journal
Format: 3
Amount: 800.25
Date: 12/25/96
Desc: Hum de dum!

//End of File
```

Each year, the illegal use of software consumes nearly 50% of your potential revenues. With the flames of piracy eating away at your profits, can you afford not to protect your software?

Software Obtained Illegally, by region, 1993 vs. 1994



HASP® is widely acclaimed as the world's most advanced software protection solution. Since 1984, thousands of leading developers have used over two million HASP keys to protect billions of dollars worth of software.

Based on a full-custom ASIC, the new HASP key redefines industry standards of security, reliability and compatibility. And because it's the smallest software protection key in the world, it's also easier on your clients!

Today, more developers are choosing HASP than any other software protection method. To learn why, and to see how easily you can increase your revenues, call now to order your HASP Developer's Kit.

1-800-223-4277

http://www.aks.com

ALADDIN

The Professional's Choice

North America — **Aladdin Software Security Inc.**
Tel: (800) 223 4277, 212-564 5678
Fax: 212-564 3377
E-mail: hasp.sales@us.aks.com

Int'l Office — **Aladdin Knowledge Systems Ltd.**
Tel: 972-3-636 2222, Fax: 972-3-537 5796
E-mail: hasp.sales@aks.com

United Kingdom — **Aladdin Knowledge Systems UK Ltd.**
Tel: 01753-622266, Fax: 01753-622262
E-mail: sales@aldn.co.uk

Japan — **Aladdin Japan Co., Ltd.**
Tel: +81 426-60 7191, Fax: +81 426-60 7194
E-mail: aladdinj@po.iijnet.or.jp

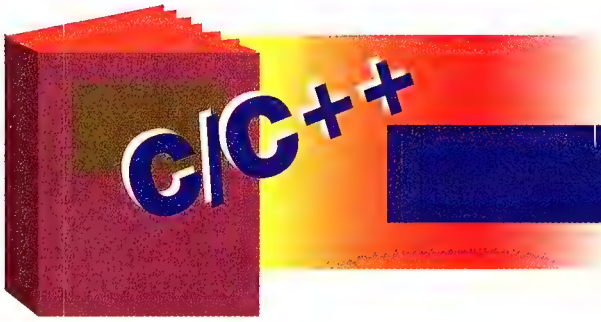
yes it runs with NetWare

Available and Ready for OS/2

Microsoft Windows Compatible

Mac OS

■ **Aladdin Benelux** 024 641 9777 ■ **Aladdin France** 1 4085 9885 ■ **Aladdin Russia** Tel 095 923 0588 ■ **Australia** Conlab 3 98985685 ■ **Chile** Micrologica 2 222 1388 ■ **China** Shanghai URI 021 6437 7828 ■ **Czech** Atas 2 766085 ■ **Denmark** Berendsen 39 577316 ■ **Egypt** Zeineldein 2 3604632 ■ **Finland** JD-Systems 0 870 3520 ■ **Germany** CSS 201 278804 ■ **Greece** Unibrain 1 6756320 ■ **Hong Kong** Hastings 02 5484629 ■ **India** Solution 11 2148254 ■ **Italy** Partner Data 2 26147380 ■ **Korea** Dae-A 2 848 4481 ■ **Mexico** SiSoft 5 2087472 ■ **New Zealand** Training 4 5666014 ■ **Poland** Systherm 61 480273 ■ **Portugal** Futurmatica 1 4116269 ■ **Romania** Interact v 64 153112 ■ **South Africa** D Le Roux 11 886 4704 ■ **Spain** PC Hardware 3 4493193 ■ **Switzerland** Opag 61 7169222 ■ **Taiwan** Teco 2 555 9676 ■ **Turkey** Mikrobeta 312 467 0635



Two STL Books

reviewed by Warren Young

Introduction

Nothing beats reading a good book for learning a new technology, such as C++'s Standard Template Library (STL). I've read two good books on STL recently, and they both have their selling points. Hopefully, you can afford to get both; but if you can't, this comparative review may help you decide between the two.

Nelson's Book

This book covers the philosophy, use, and intimate details of Hewlett-Packard's (HP) STL implementation. This is a large book, and as with any other large book, one of the first questions I ask is, "Does it really need to be that big?" In this case, I say "Yes." The reason is that, in my view, Nelson's book is really three distinct, complete, and mutually complementary books in one. The first part is actually a long essay (80 pages) which gives an overview of STL. The second is a standard tutorial/reference combination. The third part reproduces the HP STL Specification. In light of STL's almost experimental nature, this last part is closer to being a necessity than just a nice touch. Below, I review each of the three "sub-books" separately.

The book devotes much space to exploring the inner workings of STL, so you really need a good foundation in C/C++ to make sense of it easily. Beginners may still want to read this book, though, for three reasons: 1. STL tutorials are still scarce; 2. Beginners can ignore these explorations and still learn the essentials of STL with this book; and 3. Any beginners reading this magazine and interested in STL have a good chance of advancing beyond the beginner stage, so the book will eventually become more useful to them.

Warren Young is a software engineer for Educational Technology Resources, Inc. He has been programming and playing with computers since the mid-80's. He has some web pages about programming (including STL) at `http://www.cyberport.com/ \ ~tangent/programming`. You can send him email at `tangent@cyberport.com`.

Book 1: An STL Overview

The first "book" presents an overview of STL, including its history and philosophy.

Title:	*C++ Programmer's Guide to the Standard Template Library*
Author:	Mark Nelson
Publisher:	IDG Books Worldwide, 1995.
Pages:	875, softcover, with disk
Price:	$49.99 US, $69.99 Canadian
ISBN:	1-56884-314-3

Title:	*STL Tutorial and Reference Guide; C++ Programming With the Standard Template Library*
Authors:	David R. Musser and Atul Saini
Publisher:	Addison-Wesley Publishing Company, Inc., 1996.
Pages:	400, hardcover
Price:	$35.95
ISBN:	0-201-63398-1

Instead of providing the usual quick recap of well-known history, Nelson first thoroughly examines the history that led up to the creation of STL. He starts by discussing the limited genericity possible in C, works his way through C++'s evolving feature set, and ends up at C++'s recent features that make a truly generic and fast library like STL possible.

He then discusses templates. If you already understand C++ templates, you can safely skip this part. For those who can't skip it, rest assured that it's a top-notch discussion of templates as they apply to STL.

Finally, Nelson gives a nickel tour of STL itself, touching on all of STL's major features.

This first book can stand alone to serve as an "STL tasting booth" for those who still haven't decided that STL is something worth learning. If you already know you want to learn STL, this part makes a nice preview for the rest of the book.

Book 2: The STL Tutorial and Reference

The second section is what we might have expected the entire book to be: first a tutorial, then a well-written reference to STL. Indeed, some STL works dedicate the whole book to tutorial and reference. But this second "sub-book" is actually a cut above the rest of the pack. I discuss the tutorial and reference parts in turn.

The tutorial starts with the major sequential container classes: `vector`, `deque`, and `list`, as well as the container adaptors that can make a sequential container look like a stack, queue, or priority queue.

An average book would simply have shown how to use the containers by way of a few examples. What makes this a great book is that after it shows you how to use the container, it shows how HP's STL version really works. Nelson illustrates his explanations with snippets from the HP STL source code as well as clear diagrams.

One of the great things about STL is that it implements each major container in a different way, so that each has its unique performance properties. Accordingly, the next two chapters unveil the very different inner workings of the `deque` and `list` containers.

With this detailed treatment the reader can appreciate the underlying mechanisms, which is helpful in making design decisions. It's certainly nice to know that `deques` perform well when adding elements to both ends, while vectors are only quick for tail insertions. But knowing *why* these things are so gives you a sure feeling when making design decisions. Nelson puts the topping on this sundae of

GIVE YOUR APPLICATION THE CREDIT IT DESERVES!

"Making a Difference in Transaction Processing"

From Smoky Mountain Technologies...The Multilane Solutions Company

Smoky Mountain Technologies offers a wide variety of Software and Hardware options:

- Universal Payment Software as a stand-alone module or as a 'kernal' to be included with your application.
- Universal Master UPA as a back room controller to handle communication with the payment processor; transaction storage; protocol conversion; LAN controller
- More!

UniPay
UNIVERSAL PAYMENT PRODUCTS:
Universal Payment Adapter Master®
Universal Payment Software®
Universal Terminal Software®
Universal Payment Host®
Universal LAN Adapter®

PRODUCTS

MARKETS

SMOKY MOUNTAIN TECHNOLOGIES, INC.

A UniComp Company

205 Highway 64 West
Murphy, North Carolina 28906

- ☑ DOS
- ☑ Windows
- ☑ Unix

We Provide Payment Solutions for:

- Software Developers
- Cash Register Dealers
- POS Payment Providers
- Anyone Selling into the Multilane Market

Please call for more information: (704)837-6079 or Email: info@smokymtn.com

Easy Interface • Developer's Kit • Unlimited Phone Support • Installed in Thousands of Locations • Currently Used by Many Developers

Request Reader Service #127

sequential container information by pitting all three containers against one another. This "showdown" of containers demonstrates the space and time tradeoffs clearly.

The next chapter briefly covers STL's container adaptors. Adaptors give a container an interface more appropriate to the task at hand. One common use is turning a `vector` into a `stack`, for example.

The next chapter covers STL's four associative containers. As in previous chapters, Nelson covers the implementation underlying the containers, which in this case is a red-black tree. While the explanation is useful, I wonder if the book wouldn't be improved by covering the interface and use of the red-black tree class itself. It seems to me that users may find themselves wanting to derive their own classes from STL's version instead of writing their own.

The next two chapters in the second section cover allocators and iterators, which provide uniform memory management and access to the containers. These chapters cover the subject well, but I can't say it's one that is all that exciting to me. These chapters aren't boring, but aside from being as complete as the rest of the book, they aren't otherwise noteworthy. Maybe those wanting to do things like database and special memory access through custom allocators and iterators will find this chapter more indispensable.

The next chapter gives a very brief overview of STL's algorithms. It basically just tells you what's available so you can find more information in the reference section. Last but not least, Nelson discusses function objects. These clever object-oriented replacements for function pointers are key to many portions of STL.

The final chapters of the second section are a reference for STL, and as far as I've been able to tell, cover everything in HP STL. In addition to expanding upon the tutorial's topics, the reference covers sorting and searching, as well as set, heap, sequence, basic numeric, and miscellaneous operations.

Book 3: The STL Specification

This final "book" is a reprint of the original HP STL specification. This section is also quite valuable; it gives you an "STL bible" to refer to when your questions and problems outstrip the rest of the book's scope.

Criticisms and Kudos

Although I really like the book, I should point out a few of the book's faults and tradeoffs. The first tradeoff to keep in mind is that (probably due to timing issues) the book covers HP STL, which the pages of this magazine have shown to be very different from the emerging draft Standard STL. A serious fault is that the text rarely refers to the STL header files, so it's harder than it should be to find out which headers to include to get particular STL facilities.

The book more than makes up for these problems, however, with some really nice touches that make it truly great. For example, Nelson lists what each of the containers require of their contained objects. Documentation of this sort is dear to me, as I've frequently been forced to do without it.

As another nice touch, Nelson hints at many possibilities for extending STL through objects such as custom iterators and allocators that use something other than the heap for storage. If you wonder what this non-heap storage might be,

Nelson provides some examples: database access and "special" memory access like MS-DOS's EMS memory.

Nelson minimizes the book's bulk through a form of reuse: he uses a single code example to illustrate a series of points in turn.

Finally, the chapter on associative containers attempts to shed some light on why STL's creators chose to implement associative containers as red-black trees instead of hash-based structures. His argument may not end the debate, but he does illuminate the issues well.

Other Considerations

As its back cover indicates, the book focuses on PC compilers, but readers should find it easy to identify and compensate for its few minor platform dependencies. For example, the code samples each have a commented `#define` at the top to make them compile with Borland C++ 4.x. These are easily removed. Another PCism: when the book displays pointers, they are clearly of the 16-bit, "short" variety. Beyond these sorts of things, I believe the code should be portable to any STL-compatible compiler (with the caveat that the diskette is MS-DOS-formatted.)

Musser and Saini's Book

As I hinted in the previous review, I expect a book of this kind to consist of two main parts, a tutorial in the basic-to-intermediate material first, and a complete reference second. Musser and Saini follow this well-established formula.

As you might expect from its (hard) cover, this book takes a more formal approach than Nelson's. It's certainly not from the ivory tower, but this book has the standard properties of a formal work: no light conversational humor in headings and text, lots of Big-O notation, and platform independent discussions. If it had a lighter writing style, it might be easier to read, but more formal writing has its advantages, too. For example, it's easier to skim the book by glancing at headings, because you don't have to mentally translate their meanings.

The book does not attempt to teach the use of templates, so readers will need to know how to use others' templates before tackling this book.

A note on accuracy: authors are only human, so they make mistakes. Addison-Wesley recognizes this and thus makes its errata sheets easily available. The errata sheet for this book is available on the Web at `http://www.cs.rpi.edu/~musser/stl-book-errata.html`.

The Tutorial

Musser and Saini get right to their subject. Unlike Nelson, they don't try to "sell" you on STL first; if you bought the book, they assume that you really do want to learn STL. They also don't bother recounting the converging histories of STL and C++. I give Musser and Saini points for a better overview of STL, though — they provide less trivial examples and more of them.

The tutorial is one place where the book's theoretical outlook becomes obvious: the first tutorial chapter is on iterators, the glue that binds everything else together. Notwithstanding my lack of excitement about iterators (see previous review) I think this may be a better strategy than Nelson's. Without a good understanding of iterators, fitting STL algorithms and containers together can be quite frustrating.

The next chapter is on STL's algorithms. It might seem more prudent to cover containers next — after all, how can you use algorithms without containers? But you can illustrate a lot of algorithms with simple vectors, which is what the authors do. Also, the basics of vectors were covered in the introductory material. The chapter actually covers all of STL's algorithms well, and with each algorithm (or group of closely related algorithms) it includes a well-chosen example program as illustration.

Next come STL's sequential containers, all in one chapter. This arrangement makes sense because the three sequential containers belong to a "family of abstractions," which lets the authors describe only once a feature common to all of the containers. The next chapter covers associative containers in much the same manner.

The next four chapters quickly cover function objects and the various adaptor types. It is interesting to note that this book considers the adaptors separately from the facilities they adapt, whereas Nelson deals with them side-by-side.

The Example Programs

Again, Musser and Saini outdo Nelson in the examples category: each of the next four chapters is built around a largish example (as compared to the rest of the text's examples) that solves a problem by using STL. We usually like to read an example that does a lot for its size, and the authors succeed handily in creating such an example here. All of the programs are also based on a common theme, working with anagrams; as a result, they all hang together well, and between programs improvements follow a natural progression.

The next chapter focuses on defining an iterator class. This is a nice, mildly evangelistic touch on the authors' part — evangelistic because they're (naturally) trying to prove the extensibility of STL.

They then take a quick step out of the classroom and into the "real world." Here the authors discuss some of the how-to issues that come up when trying to get STL to do some nonobvious things. One of the examples show how to reduce the template code bloat problem with a novel application of polymorphism.

The Reference

The book's reference section is, like the rest of the book, quite "standard." It is divided into chapters roughly parallel to the tutorial's, and each subsection discusses the details of a particular facility. I didn't find it lacking in any way. One thing covered in this section that isn't covered in the tutorial are allocators. The reference provides enough information to let you write your own allocators if the necessity presents itself.

Criticisms and Kudos, Part 2

I do have a criticism of the book. I'm really unimpressed with the illustrations in this book. They're not inaccurate, but they just don't communicate their information as well as Nelson's do. Fortunately, I think this book doesn't need illustrations as much as Nelson's book does, so it doesn't suffer all that much. I still think that they could have been clearer, though.

Now here are some nice touches: though the rest of the book uses the HP reference implementation of STL in its examples, the introduction to the book does explain the header changes that the ISO C++ committee made. These particular changes will probably cause some headaches, and this feature should help alleviate the suffering.

The tutorial chapter on iterators offers a table of containers paired with the iterators they return. This makes matching up containers with compatible algorithms easier.

The chapter on associative containers doesn't spend much time arguing for red-black trees over hash tables, but the book does mention that there are STL-based hash containers available from the main STL site (`ftp://butler.hpl.hp.com/stl`), which was a nice revelation. The other sites mentioned in the same appendix are also worth a visit, and their inclusion is yet another nice touch.

Comparing the Two Books

These two books differ in philosophy, in a couple of fundamental ways. For one thing, Nelson's book tries to do more than Musser and Saini's does. Besides the reprint of the STL spec, the main extra Nelson offers is a running exploration into the workings and design of an implementation of STL. However, in doing so, he constrains the book somewhat to one implementation and one platform.

On the other hand, Nelson's coverage gives us an insight into STL that, once obtained, is hard to imagine doing without. This added insight gives the designer in all of us information to help us choose between similar STL facilities, as well as some good lessons in design. Plus, it's really easy to step past the dependencies if they get in your way. In all likelihood, future implementations of STL will look much like the HP reference implementation, and the PC dependencies are easy to spot and fix.

It all adds up to this: I think Nelson's book is a great introduction to the library, and that it provides an insight into some of the "Zen" of STL, which Musser and Saini's book does not. Also, Nelson's inclusion of the STL spec and a decent reference makes his book useful beyond the initial learning period.

Does that mean the other book is useless? No, not by any means. It's just *different*. For one thing, I think that Musser and Saini have done a better job of organizing their book. The organization of Nelson's book is okay, but I get the impression it wouldn't be as easy to use as a reference. By better organization I mean that it is easier to predict where to find coverage of a certain feature in Musser and Saini. In particular, Musser and Saini make a clear separation between the straight reference material and the tutorial material. Of course, Nelson's organization has its points as well: his reference section contains examples as well as the tutorial, which is nice when you're still learning how to use something and need both reference and example material at the same time.

I mentioned that Nelson's book covers a working implementation of STL. I personally consider this a plus, but perhaps not everyone will. If you don't have the time or inclination to understand the details of an STL implementation, you may be better off with Musser and Saini.

The Verdict

I think Nelson's book makes a better tutorial, and that Musser and Saini have the better reference.

If you're a non-PC user who can't stand PCisms, stay away from the Nelson book. If you're more interested in the theory of STL, start with Musser and Saini's book.

I'd really hate to choose between the two books. But if you must, I'd recommend doing what I did, if you can: learn with Nelson, and if you decide you need better reference or how-to material, get the Musser and Saini book. ❑

Instant Access to Six Years of C/C++ Users Journal

Stop fumbling through your magazine heap

The CD's easy-to-use interface gives you powerful, full-text search capability. Find the articles you need instantly.

Never buy another back issue

It's all here: whether you began subscribing last month, last year, or at the beginning of the nineties, you'll have six years of *C/C++ Users Journal* at your command.

Don't reinvent the wheel

C/C++ Users Journal articles supply code that you can drop right into your current application. From tiny functions to class definitions, fellow programmers share the fruits of their labor with you.

The *C/C++ Users Journal CD-ROM* contains six years' of the leading C/C++ programming magazine. Hundreds of articles, most with source code, provide practical technical information on debugging, algorithms, class libraries, image processing, portability, optimization, real-time and embedded systems, ANSI C and C++ standards — and much more.

Order Today!

(800) 444-4881

Call 24 hours a day — 7 days a week
Or complete the form and return to :

FAX: (913) 841-2624
E-MAIL: orders@mfi.com
INT'L: Use mail, fax, e-mail, or call (913) 841-1631

FULL MONEY BACK GUARANTEE. If you don't find the *C/C++ Users Journal CD-ROM* worth the price, we'll return your money – no questions asked.

Please send me a *C/C++ Users Journal CD-ROM* today! My price is $49.95 (plus shipping: $2.00 U.S./Canada; all other countries $10.00) Sales tax added in the following states: CA (8.5%), GA (6%), IL (6.25%), KS (6.9%), NY (8.25%), TX (8.25%).

Name _____

Address _____

City _____ State _____ ZIP _____ Country _____

Phone _____

❑ Check ❑ VISA ❑ Mastercard ❑ Amex

Credit Card#: _____ Exp. Date _____

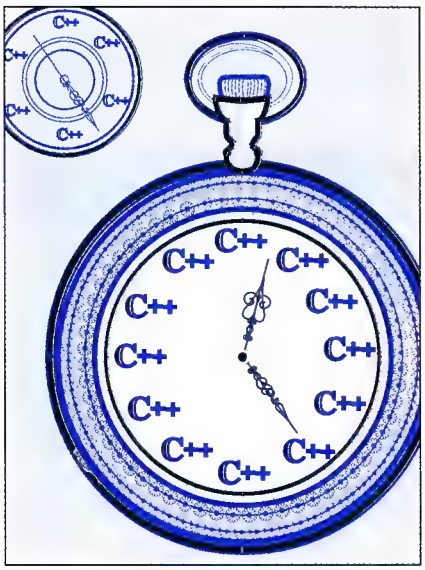
Signature _____

C/C++ Users Journal CD-ROM
1601 West 23rd Street, Suite 200, Lawrence, KS 66046-2700

Greg Messer

A C++ Chronograph Class

Here's a class for timing program intervals that's powerful in its simplicity.



A Programming Problem Solved

One coding task I find myself doing over and over in my programs is timing. I time the program as a whole, I time the individual phases of the program, I time the I/O throughput, and I estimate the program's completion time. That timing code is just complicated enough to make me think carefully while writing it — and just tedious enough to make me want to never write it again. What I need is an easily reusable, plug-in tool to simplify those timing tasks for me.

To fill that need I have created a C++ `Chronograph` class, a software stopwatch to handle my repetitive timing requirements. The `Chronograph` class provides the same functionality as a good digital stopwatch. It will give the elapsed time, split times, and lap times. You can start, stop, restart, and reset the `Chronograph`. It can break out the hours, minutes, and seconds for you. Best of all, several `Chronograph`'s can run at the same time without slowing down your program one bit.

`Chronograph` Functionality

The `Chronograph` can be in one of two modes, running or stopped. `Chronograph::start` does one of two things, depending on whether the `Chronograph` is running or stopped. When running, `Chronograph::start` will restart the `Chronograph` at zero and leave it running. When stopped, it restarts the `Chronograph` from zero. I coded `Chronograph::start` to work this way because that is how a stopwatch works. The return value of this member function is the elapsed time.

Calling `Chronograph::elapsed` or `Chronograph::split` gets the elapsed time of execution up to the current moment. The elapsed time is the amount of time since the `Chronograph` was started. Splits are the same as the elapsed time, so function `split` just calls `elapsed`.

`Chronograph::elapsedHMS` breaks down the elapsed time into hours, minutes, and seconds. Using this function allows you to report "5 hours, 32 minutes, 10.25 seconds" instead of "19930.25 seconds".

Laps are different from the simpler elapsed time. If you call `Chronograph::lap` it will return the amount of time since `Chronograph::lap` was last called. Laps are the periodic intervals that exist within a program. Like a runner doing laps on a track, programs loop and perform other operations that have distinct starting and stopping moments. For example, you can keep track of the time spent processing 1,000 database records, so you can estimate the program's completion time, or keep track of the time spent in each phase of a multi-phase process.

`Chronograph::stop` puts the `Chronograph` in the stopped mode. The `Chronograph` stores the clock value when it was stopped and uses it to adjust the start time at the next restart. The amount of time spent in the stopped mode does not count toward the elapsed time. The return value of this member function is the elapsed time. If you call one of the elapsed member functions `Chronograph::elapsed` or `Chronograph::elapsedHMS` while the Chronograph is stopped, you will get the elapsed time when the Chronograph was stopped.

I have provided `Chronograph::isstopped` to indicate whether the `Chronograph` is stopped. It returns 1 if the `Chronograph` is stopped, and 0 if it is running. I recommend changing this member function to use the new `bool` type when it becomes available with your compiler.

Implementation Details

How can you have several timers going at once and not slow down your program? Well, you don't do it by installing hardware interrupt handling routines, although those might provide a higher resolution. You give up some resolution by simply using the timer that is already in place — the system clock.

The ANSI C `clock` Function

The `Chronograph` class uses the ANSI C `clock` function to get the time. The `clock` function returns a value of the type `clock_t`, which is the number of clock ticks since a program started. To calculate the number of seconds represented by those clock ticks, divide the `clock_t` value by the ANSI C macro `CLOCKS_PER_SEC`.

The `CLOCKS_PER_SEC` macro will likely be different from platform to platform, but the

Greg Messer works for Systems Integration and Imaging Technology, Inc. as a programmer/analyst and system administrator. He started programming 12 years ago on mainframes and now specializes in high performance client/server programming on networked PCs. Greg can be reached by e-mail at `73510.1430@compuserve.com`.

equations that use it will remain the same. The CLOCKS_PER_SEC macro in Borland C is defined as 1000.0. Borland's clock function therefore returns the thousandths of a second since a program started. The MS-DOS system clock that Borland C uses in its clock function is not really accurate to the thousandth of a second, but the Chronograph class is not meant to be that accurate either. The MS-DOS system clock is more than accurate enough for the long-duration timing tasks I had in mind.

Listing 1: *Timing execution of an entire program*

```
#include <iostream.h>
#include <string.h>
#include <ctype.h>
#include "chrono.h"

//allow long input lines
#define MAXLINE 1024

void getcmt(istream &in);
void usage(void);

// Global program name, for messages.
char *pgm = "GETCMT";

// ----------------------------------------
main(int argc, char *argv[])
{
   int    i;
   double elapsed_time;

   // This Chronograph measures total
   // elapsed time of program.  It starts
   // running as soon as it's instantiated.
   Chronograph chrono;

   // ----------------------------------------

   // Command line processing, details
   // omitted
   // ...

   // ----------------------------------------

   getcmt(cin);

   elapsed_time = chrono.elapsed();

   // Show the total time this program was
   // running.
   cout << endl;
   cout << pgm << " main()";
   cout << ": Total elapsed time: ";
   cout.precision(2);
   cout <<  elapsed_time;
   cout << " seconds." << endl;
   cout << endl;

   return 0;
}
//End of File
```

The operating system maintains a clock tick counter at all times, and the initial value of that counter is stored by the startup code before main is called. Subsequent calls to clock query the OS clock value, then subtract the value stored at startup. The result is the number of clock ticks since the program started.

This method of time measurement has little effect on your program's execution speed, regardless of how many Chronographs you have instantiated. The only time the Chronograph affects the speed of your program is when you call one its member functions. The OS is servicing the clock ticks for you, whether you use them or not.

Simple Calculations

The simple calculations used by the Chronograph class depend on a few private variables. By keeping track of the clock values when the Chronograph was instantiated, when the Chronograph was stopped, and when the lap member function was last called, the Chronograph calculates elapsed

and lap times as the difference between two clock values.

A private member function, `Chronograph::diff`, accepts two `clock_t` values and returns the difference between them in seconds. This function is used whenever the elapsed or lap times need to be calculated.

Handling the Modes

As mentioned earlier, the Chronograph can be in one of two modes, running or stopped. If the Chronograph is ever stopped, it can be restarted or it can be reset and then started from zero. The elapsed and lap times can be queried regardless of the Chronograph mode. Being able to stop the Chronograph allows you to keep sections of code from being timed. For instance, you can stop the `Chronograph` while waiting for user input, then restart it when your program goes back to work. The time spent waiting for the user will not affect your timing statistics.

A Simple Demonstration

I show part of a filter program to demonstrate the use of the `Chronograph` class. The program is named `GETCMT`. It reads a C or C++ source file from the standard input stream and writes the comments to the standard output stream. I use this utility to give me a starting point for documenting my source code.

A simple example of the `Chronograph` use occurs in the `main` function of `GETCMT.CPP` (Listing 1). The `main` function uses this `Chronograph` to time the entire program. The `Chronograph` is instantiated a few lines down from `main`'s parameter list. It hits the ground running, so to speak. You don't need to call any member functions to start it. The program takes the elapsed time near the end of `main`, after calling the function `getcmt`. Those two lines are all that are required to time an entire program using the `Chronograph`.

Use `setprecision(2)` to set up your output stream before you display the elapsed time returned by the `Chronograph` member functions. If you use `printf` for output, then use "`%.2f`" as your format string.

A Complex Use of the Chronograph

The `getcmt` function demonstrates a more complex use of the `Chronograph` class. This function uses three `Chronographs` — one to time the function as a whole, one to measure the time spent inside of comments, and one to measure the time spent outside of comments. Listing 2 shows portions of `getcmt`, with representative `Chronograph` member function calls. (Full source code is available electronically — see p. 3 for details.)

The three Chronographs are instantiated after creation of an I/O buffer and some auto variables. Since `getcmt` has yet to encounter any comments upon startup, the `inside` Chronograph is reset to zero shortly after instantiation, and it remains stopped. The two comment Chronographs (`inside` and `outside`) stop and start when

Listing 2: *A function that uses three chronographs*

```
void getcmt(istream &in)
{
   int  i, j, len;
   long linecount = 0l;

   double hours, minutes, seconds;

   // Flag: 0 = not in comment,
   //       1 = in C comment
   //       2 = in C++ comment

   int incomment = 0;

   // Flag: 0 = no comment on this line
   //       1 = comment on this line

   int commentline = 0;

   char inbuff[MAXLINE + 1]; // I/O buffer

   double elapsed_time;
   double elapsed_time_inside;
   double elapsed_time_outside;

   // One Chronograph to count the time
   // spent in this function.

   Chronograph chrono;

   // Two Chronographs, to count the time
   // spent inside and outside of comments.

   Chronograph inside;
   Chronograph outside;

   // -----------------------------------

   // reset the "inside comment" timer to
   // zero and stop it

   inside.reset();

   cout << endl;

   in.getline(inbuff, MAXLINE);

   while(in.eof() == 0)
   {
      linecount++;

      //demonstrate use of lap time
      if(linecount % 100 == 0)
      {
         cout << pgm;
         cout << " getcmt()";
         cout << ": 100 lines in ";
         cout.precision(2);
         cout << chrono.lap() << " seconds.";
         cout << endl;
      }

      len = strlen(inbuff);
      i = 0;

      // while chars are left on the line...
      while(i < len)
      {
         // process depending on if we are
         // in a comment and what type it is
         switch(incomment)
         {
            case 0:
               // not inside a comment
               // C comment detection
               if(inbuff[i] == '/' &&
                  inbuff[i+1] == '*')
               {
                  outside.stop();
                  inside.start();

                  incomment = 1;
                  commentline = 1;

                  i += 2;
               }


               // C++ comment detection
               // details omitted
               // ...

            case 1:
               // inside a C comment

               // Nested C or C++ comment
               // detection details omitted
               // ...

               // end of C comment detected
               else if(inbuff[i] == '*' &&
                       inbuff[i+1] == '/')
               {
                  inside.stop();
                  outside.start();

                  incomment = 0;

                  i += 2;
               }
               else
                  i++;

               break;

            case 2:
               // inside a C++ comment
               // details omitted
               // ...

            default:
               // error, bad "incomment"
               // shouldn't happen
               cerr << "Error!" << endl;

            return;

         }
         // end of switch()
      }
      // end of while()

      if(commentline == 1)
         cout << inbuff << endl;

      if(incomment == 1)
         commentline = 1;
      else
         commentline = 0;

      in.getline(inbuff, MAXLINE);
   }

   // stop all three of the timers
   // and get the elapsed times
   elapsed_time         = chrono.stop();
   elapsed_time_inside  = inside.stop();
   elapsed_time_outside = outside.stop();


   // show times spent inside and outside
   // of comments.  Details omitted
   // ...


   return;
}
//End of File
```

Listing 3: *The Chronograph class*

```
#ifndef CHRONO_H
#define CHRONO_H

#include <time.h>

class Chronograph
{
private:

   // when the Chronograph was started
   clock_t _start;

   // when the Chronograph was stopped
   // If _stopped != 0, the Chronograph
   // is stopped.
   clock_t _stopped;

   // when the last lap was read
   clock_t _lastlap;

   // the last elapsed time taken
   double _elapsed;

   /*
    diff() calculates the difference IN
    SECONDS between two clock_t values.
    */

   double diff(clock_t start, clock_t end)
   {
      return (end - start) / CLOCKS_PER_SEC;
   }

public:

   // default constructor
   Chronograph()
   {
      reset();
      start();
   }

   // destructor
   ~Chronograph()
   {
   }

   double start(void)
   {
      // current clock
      clock_t ct;

      // how long the Chronograph was stopped
      clock_t ctd;

      ct = clock();

      if(ct == 0)
         ct++;

      if(_stopped)
      {
         ctd = ct - _stopped;
         _start += ctd;
         _lastlap += ctd;

         _stopped = 0;

         return elapsed();
      }
      else
      {
         _lastlap = ct;
         _start = ct;
         _elapsed = 0.0;

         return _elapsed;
      }
   }

   double stop(void)
   {
```

getcmt's scanning passes into and out of comments.

When getcmt detects the start of a comment it executes the following statements:

```
outside.stop();
inside.start();
```

Likewise, when getcmt detects the end of a comment, it executes the following statements:

```
inside.stop();
outside.start();
```

That is all the code needed to maintain those two Chronographs. The simplicity of the interface is what allows you to easily add one or more Chronographs to an existing program.

getcmt stops all three Chronographs just after the end of its main while loop. Calling Chronograph::stop stops a Chronograph and also returns elapsed time.

Although not shown here, the full source code version of `getcmt` calls member function `Chronograph::elapsedHMS` to get the total time broken out as hours, minutes, and seconds. The sample program most likely will not need to measure hours and minutes, but I have some programs in daily production that use the long-range capabilities of the `Chronograph`. The implementation for `elapsedHMS` is available electronically with the full source code (see p. 3).

Other Uses for the Chronograph

The `Chronograph` has some other uses besides simple time measurement. For example, let's say you are using `Chronograph::lap` to track how long you spend on some cyclical aspect of your program, such as processing database records. If you know the total number of those records in advance you can call `Chronograph::lap` after each 1,000

Listing 3: *continued*

```
    if(!_stopped)
    {
       _stopped = clock();

       if(_stopped == 0)
          _stopped++;
    }

    return elapsed();
}

double reset(void)
{
    clock_t ct;     // current clock

    ct = clock();

    if(ct == 0)
       ct++;

    _lastlap = ct;
    _start = ct;
    _stopped = ct;
    _elapsed = 0.0;

    return _elapsed;
}

/*
 lap() returns the number of seconds
 since lap() was last called.
 */

double lap(void)
{
    clock_t ct;     // current clock
    double dl;      // lap time

    if(_stopped)
       ct = _stopped;
    else
    {
       ct = clock();

       if(ct == 0)
          ct++;
    }

    dl = diff(_lastlap, ct);

    if(!_stopped)
          _lastlap = ct;

    return dl;
}

double split(void)
{
    return elapsed();
}

int isstopped(void)
{
    if(_stopped)
       return 1;
    else
       return 0;
}

/*
 elapsed() returns the elapsed time in
 seconds that the Chronograph has been
 running.
 */

double elapsed(void)
{
    clock_t ct;     // current clock

    if(_stopped)
       ct = _stopped;
    else
```

Your Roadway to Building Large Scale Systems

C++ WORLD

The Users Conference and Exhibition

Now In its 8th Successful Year!

The Mecca for C++ users.

Over 2,000 serious C++ programmers will converge in Dallas on Nov. 11-15. It's the largest C++ event of the year. C++ World is created for C++ users on all platforms and levels. Meet face-to-face with the top experts and best-selling book authors. Expand and hone your C++ programming skills.

NEW and IMPROVED tracks...

Each offering vital education opportunities for the C++ programmer. Choose from:

- Methods, Analysis and Design
- Components and Frameworks
- Persistence
- Distribute Objects
- Software Development
- Large Scale Design

PLUS: a special "Gurus Only" accelerated track specifically designed for advanced programmers with 3+ years C++ experience.

Past and Present Exhibitors (partial list):

Addison Wesley • Alsys • Borland • *C++ Report* • *C/C++ Users Journal* • Cayenne Technologies • CenterLine • Datalink • DevelopMentor • Expersoft • IBM • ICON Computing • IEEE Computer Society • InterSolv • *JOOP* • *Java Report* • McCabe & Associates • Metrowerks • MultiQuest • Neuron Data • Object Design • Object International • *Object Magazine* • ObjectSpace • O'Keefe & Assoc • Pencom • POET Software • Prentice Hall • ProtoSoft • Pure Software • Rogue Wave • Semaphore • SIGS Publications • Software Emancipation • Symantec • TakeFive Software • Teknekron Comm. Systems • TRW • UniSQL • VERITAS Software • *The X Journal* • XVT Software • Zinc Software

What You Will Learn:

Build on your C++ foundation through a variety of educational classes — over 40 to choose from. You'll learn:

- How to select a distributed object technology path by comparing and contrasting ActiveX, CORBA and OpenDoc.
- How to move from C++ to Java™.
- The secrets to reuse.
- How to get the most out of Visual C++ and MFC.
- How to design and manage class libraries.
- Multithreading tips in C++.
- How to decouple and leverage code.
- Effective exception handling.
- Testing & debugging tips.
- Understand and use the standard template library.
- How to use modular design with C++.
- Managing pointers and complexity in C++.
- Using patterns effectively.
- Using ODBMS with C++.
- The Unification of methods.
- Web programming tips and techniques.

November 11-15, 1996

Grand Kempinski Hotel
Dallas, Texas

For more information
Call 212.242.7515

Reach our website at
http://www.sigs.com

Email us at conferences@sigs.com

Sponsored by:
C++REPORT

Presented by:
SIGS CONFERENCES

records and use those lap times to dynamically update an estimated completion time.

I also put the Chronograph to good use in a "spinner" class that I wrote to indicate a program's progress. By repeatedly cycling through a series of characters, ('/', '-', '\', and '|'), I display a character that appears to be spinning.

The spinner class contains a private Chronograph that makes sure the screen update does not happen too often. No matter how frequently an application tries to update the spinner, it only displays itself at one second intervals.

Conclusion

I use the Chronograph class daily. Often I am asked how long it will take to execute some program. With as little as two lines of code I can add a Chronograph to any C++ program and get the timing statistics I need. With only a few more lines of code I can create any timing statistic I can imagine.

The need for timing exists in virtually every significant program. With this Chronograph, you can add timing capabilities in a few moments and then focus on the more interesting parts of the program. That's the real power of a great tool. ❑

Listing 3: *continued*

```
        {
            ct = clock();

            if(ct == 0)
                ct++;
        }

        _elapsed = diff(_start, ct);

        return _elapsed;
    }

    double elapsedHMS(double &Hours,
                      double &Mins,
                      double &Secs);
};

#endif

// end of chrono.h

//End of File
```

Anil Hemrajani

The Java and C Connection

Sure, Java is a neat new toy. But it's nice to know you can mix in a bit of C code, from time to time, to beef up a Java program.

In recent months, one of the hottest technologies to hit the streets has been Java. It has taken the information systems industry by such a storm that most software vendors have jumped on the bandwagon to develop products for Java. Meanwhile, many companies still have large investments in applications written in other languages, such as C. My goal here is to show you how to interface to C from Java. I assume that you have a working knowledge of both Java and C.

Java provides a native method interface which can be used to invoke functions written in other languages. Currently only an interface to C is supported. (Of course, C++ source code can be accessed via C functions). There are approximately six steps required for connecting a Java program to C. The two main steps include writing the Java program and C functions. The remaining steps are required for gluing the two together.

Here is a simple example to illustrate each of these six steps. The example consists of a Java program that calls a C function to print the message "Hello Java" (a twist on the typical "Hello World" message). To keep the first example simple, no parameters are passed to the C function and no values are returned from it. Figure 1 provides a visual representation of the various components created from the following six steps required to make the Java and C connection:

1) Write a Java program.
2) Compile it.
3) Generate a header file for the C function.
4) Generate a stub file for the C function.
5) Write the C function.
6) Build a dynamic link library with the C function.

Let's look at each of these steps in more detail.

Anil Hemrajani currently provides software engineering and training consulting services to a Fortune 500 corporation in McLean, VA. He can be contacted at anil@patriot.net or via http://www.patriot.net/users/anil/.

Step 1

The following is a simple Java program that defines a native method SayHi. Its implementation is provided in step 5:

```
public class HelloJava
{
    public native void SayHi();

    public static void
        main(String args[])
    {
    System.loadLibrary("hello");
    new HelloJava().SayHi();
    }
}
```

Native methods are declared much the same way as regular Java methods, with two differences. The declaration must contain the native keyword, and there is no body for the method in Java. The body is provided in the C function. The loadLibrary method (part of the Java System class) must be called prior to using the native method so that the dynamic link library containing the C function can be loaded. The new expression must be used to instantiate a native method.

What I have just shown here is one way of instantiating a native method. Alternatively, the native method could be declared in another Java class. In that case, we instantiate that Java class using a new expression, then simply call the native method off of the class. This style is used in Listing 3 and described below.

Step 2

Compile the Java class using a Java compiler, such as javac utility provided with the Java Development Kit (JDK). Running the following command will generate a Java byte code file, HelloJava.class:

```
javac HelloJava.java
```

Step 3

Generate a header file for the C function using the javah utility. The generated header file (HelloJava.h) basically provides a prototype for the C function, which is covered in step 5.

```
javah HelloJava
```

Step 4

Generate a stub function, using the javah utility, to connect the Java class to the C function. The following command will generate a stub file named HelloJava.c:

```
javah -stubs HelloJava
```

Step 5

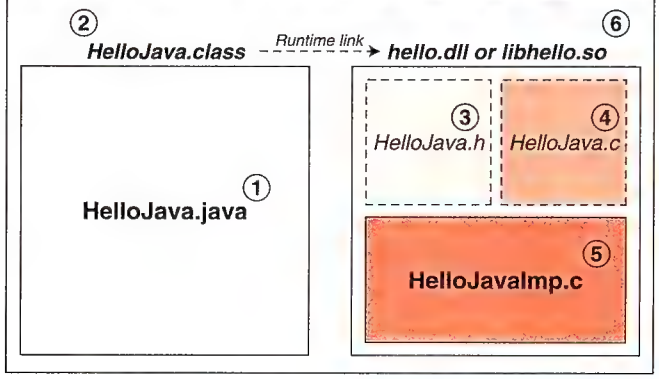
Provide an implementation for the C function, SayHi:

```
/* HelloJavaImp.c: Implementation of
    Java native method, SayHi(). */
#include "HelloJava.h"
#include <stdio.h>

void HelloJava_SayHi(struct HHelloJava *javaObj)
{
    printf("Hello Java!\n");
}
```

Three things in the above source code deserve mention. First, the header file generated in step 3, `HelloJava.h`, must be included in the C module. Second, notice the name of the C function, `HelloJava_SayHi`. It contains the name of the Java class and the native method separated by an underscore. You will come across this naming convention (`class_method`) whenever you are interfacing with native methods in Java.

Last, an automatic parameter is always passed as the first parameter to a native method. It is essentially a handle to the Java class invoking the native method. You can think of this parameter as the `this` keyword in C++. I discuss how to use this parameter below.

Figure 1: *Using C with Java*



Step 6

Build a dynamic link library. Here is a simple makefile which contains targets for building a dynamic link library on Windows 95, using the Microsoft C/C++ compiler, and a shared library on Solaris. The environment variable `JAVAHOME` points to the root directory of the JDK:

```
all:
    @echo Please specify a target: Win95 or Sun.

Win95:
    cl HelloJavaImp.c HelloJava.c \
-I$(JAVAHOME)\include -I$(JAVAHOME)\include\win32 \
-Fehello.dll -MD -LD $(JAVAHOME)\lib\javai.lib

Sun:
    cc -G -I$(JAVAHOME)/include -
I$(JAVAHOME)/include/solaris \
HelloJavaImp.c HelloJava.c -o libhello.so
```

Once you have completed the above six steps successfully, you are ready to run the sample Java and C application. Here is how you would test the application and the output you should receive:

```
C:\ANIL\Doc\Articles\Java-C\Example1> java HelloJava
Hello Java!
```

Passing Parameters and Returning Values

Now that I have shown you the basic steps required to connect Java programs to C functions, it is time to get a little fancier by passing parameters to a C function and returning values from it. Since steps 2, 3, 4 and 6 are similar for all cases, I concentrate on steps 1 and 5 which deal with a Java program and C function(s), respectively. But before plunging into the specifics of these examples, I provide a quick overview of Java data types and how they correspond to C data types.

The following table provides a list of Java native data types and their machine sizes:

Type	Size
`byte`	8-bit
`char`	16-bit
`short`	16-bit
`int`	32-bit
`float`	32-bit
`long`	64-bit
`double`	64-bit

As you might notice, some Java data types are larger than their equivalent C data types. For example, a `char` is 16 bits in Java and

(typically) eight bits in C. The numeric data types are similar to those you find on UNIX systems, but they are typically larger than ones on MS-DOS based systems. An `int` is usually two bytes on MS-DOS, for example, but it is four bytes in Java. The sizes for Java data types are the same across all supported platforms, which makes the language more portable.

Listings 1 and 2 show a sample Java program and a sample C function (tested on Windows 95 and Solaris), respectively. The purpose of this sample application is simple. The Java class calls a C function with certain parameters. The C function prints the values in these parameters to `stdout` using `printf` and returns the number of characters printed to the Java class, which in turn prints a message indicating how many characters were printed.

Let's review some of the significant points in these examples. In the Java program, notice how I placed the native method in a separate class (`PrintInC`) from the "main" class (`ThePrinter`). I did this to show you an alternative way of invoking native methods. Also notice the `static` block in the `PrintInC` class. This is a good place to automatically perform tasks when the class is loaded, which in my case happened to be loading a dynamic library:

```
static { System.loadLibrary("print"); }
```

Now, let's consider the C implementation. First notice that I include a StubPreamble.h header file. This is a JDK include file which contains necessary structures (such as `HArrayOfInt`), functions prototypes (such as `makeCString`), and macros (such as `unhand`). Note also the mapping of data types used in Java versus what I had to use in C:

```
public native int doPrint(long l,
    int    i[],
    int    count,
    double d,
    String s);

long PrintInC_doPrint(struct HPrintInC *this,
    int64_t l,
    HArrayOfInt *ai, long iCount,
    double d,
    struct Hjava_lang_String *s)
```

Notice how each `int` has been mapped to `long`. An `int` may be only two bytes on MS-DOS but is always four bytes in Java. Similarly, a Java `long` argument has been mapped to the type `int64_t` in C. The other two notable parameters in these examples are a Java array (`int[]`) and a Java object (`String`). The Java `int[]` array gets mapped to the C structure `HArrayOfInt`.

`HArrayOfInt` is one example of a standard naming convention used in the `StubPreamble.h` header file. Other examples include `HArrayOfLong`, `HArrayOfByte`, and so on. Java objects also use standard naming conventions such as `H` for handle, the name of the language (e.g. `java`), the package (class library) containing the class (`lang`), and the name of the object (`String`). So a structure for the Java `String` object, which is part of the `lang` package, is named `Hjava_lang_String`.

Finally, let me point out a C macro (`unhand`) and function (`makeCString`) used in this example. The `unhand` macro provides access to the variables in Java's C Structures, such as the `HArrayOfInt`, as shown here:

```
i = unhand(ai)->body;
```

Another example of `unhand` is shown in the next section.

The `makeCString` function takes a Java `String` as an argument and returns a corresponding `char` pointer. There are two other functions related to conversions between Java strings and C character arrays, `makeJavaString` and `java2Cstring`. The former is covered in the next section along with an example. The latter is similar to `makeCString`, except that it works more like `strncpy` — it copies a Java `String` object into an existing `char` array as shown in the following example:

```
char OutFile[127+1];
javaString2CString(pOutFile, OutFile, sizeof(OutFile));
```

Accessing Java Objects

So far I have shown you how to call C functions from Java. Now I show you how to go the other way, accessing Java objects from C. Java provides the ability for C functions to access data members in Java objects, create instances of Java classes, and invoke dynamic and static methods in the Java class. Listings 3 and 4 demonstrate how to accomplish most of these tasks.

Listing 3 contains mostly the same type of Java source code as my previous examples, but one thing is done slightly differently and

Listing 1: Sample Java program

```
// ThePrinter:
// Call native methods to print stuff

public class ThePrinter
{
   public static void main(String args[])
   {
      int count=4;
      int  i[] = new int[count];
      i[0] = 10;
      i[1] = 75;
      i[2] = 95;
      i[3] = 115;

      int printed = new PrintInC().doPrint(
                        25,
                        i, count,
                        100.33,
                        "Hello C");

      System.out.println("Java: " +
                         printed  +
                         " chars printed");
   }
}

class PrintInC
{
   public native int doPrint(long   l,
                             int    i[],
                             int    count,
                             double d,
                             String s);

   static
   { System.loadLibrary("print");  }
}
// End of file
```

worth mentioning. The JavaAccessor class illustrates how a Java class containing multiple native methods can be instantiated and then individual methods in that class can be invoked, as shown here:

```
JavaAccessor ja = new JavaAccessor();
..
System.out.println("PATH=" + ja.getEnv("PATH"));
ja.deleteFile("dummy.txt");
```

Listing 4 requires a little more explanation, as there are several new things I have not covered yet. First, the getEnv native method returns a Java object (String), unlike the other examples I have shown so far which return native data types. Notice how you have to use makeJavaString, the opposite of makeCString, to return a Java String object:

```
return makeJavaString(value, strlen(value));
```

You can access data members from the "automatic" parameter that is passed to every native method, in this case the pointer to HJavaAccessor. You simply reference a data member in a Java class by using the unhand macro to dereference the Java object:

```
long JavaAccessor_deleteFile(struct HJavaAccessor *javaObj,
                struct Hjava_lang_String *fileName)
{
    ...
    unhand(javaObj)->delRC=rc;
```

SOFTWARE DEVELOPMENT

C++ AND WINDOWS PROGRAMMING

COMING SOON TO:

WASHINGTON, DC
May 13 – 17

NEW YORK
May 20 – 24

BOSTON
June 3 – 7

TORONTO
June 10 – 14

SEATTLE
June 17 – 21

SALT LAKE CITY
July 29 – August 2

COLUMBUS
September 9 – 13

DALLAS
September 23-27

MINNEAPOLIS
October 7 – 11

CHICAGO
October 14 – 18

DENVER
November 4 – 8

ATLANTA
November 18 – 22

Taught by Dan Saks

C++ Fundamentals Not compiler-specific; suitable for any programming platform

- Learn today's premiere programming language from a leading expert and teacher
- New! Integrated Labs to master concepts through practical programming exercises (bring your own computer or laptop)
- Progress step-by-step through the key language features
- Understand how data abstraction, inheritance and polymorphism support sound design practice
- Move from C to C++ in just 2 days
- Prepare for Windows-based programming with C++

Taught by Richard Hale Shaw

Windows 95/Windows NT Programming with Visual C++ 4.0 and MFC 4.0

- Learn the fundamentals of Windows programming with VC++/MFC 4.0
- New! Integrated Labs (bring your own computer or laptop)
- Explore the MFC architecture
- Use VC++/MFC 4.0 for Rapid Application Development
- Apply Visual techniques to build, derive, and modify new C++ classes
- Learn MFC's document-view architecture
- Build dialog-only applications
- Form views, toolbars with cool user-interface components
- Use VC++ 4.0's built-in Component Gallery objects
- Learn how to back-port to Windows 3.1
- Use OLE controls in your VC++ applications
- Create MFC extension DLLs
- Use MFC to get a jump-start on OLE
- Discover the wonder of OLE Automation
- Build Automation servers

CALL US TODAY TO FIND OUT MORE!

PHONE: (415) 905-2702
FAX: (415) 905-2222
E-MAIL: sdconfs@mfi.com

FOR MORE CLASS INFORMATION:

HTTP://WWW.MFI.COM/SDCONFS

Produced by the Software Development Conference & Show Group of Miller Freeman, Inc.
600 Harrison Street, San Francisco, CA 94107

Miller Freeman
A United News & Media Company

Listing 2: PrintInCImp.c: C functions for Java class

```
#include <stdio.h>
#include <StubPreamble.h>
#include "PrintInC.h"


/*** Java to C demo: Print data values ***/
long PrintInC_doPrint(struct HPrintInC *this,
                      int64_t l,
                      HArrayOfInt *ai, long iCount,
                      double d,
                      struct Hjava_lang_String *s)
{
   int charsPrinted=0, idx=0;
   long *i;


   /* Print value of "long" (Java int) */
   charsPrinted += printf("C: l = %ld\n", l);


   /* Print array of longs */
   i = unhand(ai)->body;
   for (idx=0; idx < iCount; idx++)
      charsPrinted += printf("C: i[%d] = %d\n", idx, i[idx]);


   /* Print value of "double */
   charsPrinted += printf("C: d = %f\n", d);


   /* Print Java string */
   charsPrinted += printf("C: s = %s\n", makeCString(s));


   /* Return total characters printed */
   return charsPrinted;
}
// End of file
```


Finally, you can instantiate Java classes and invoke methods within them. Three C functions accomplish this — `execute_java_constructor`, `execute_java_dynamic_method`, and `execute_java_static_method`.

`execute_java_constructor` (shown below) expects at least four parameters. It can require more, depending on the number of parameters a constructor requires. The first parameter indicates the language/execution environment, which for our purposes is always zero for Java. The second parameter is the name of the Java class. The third is currently reserved for future use. The fourth is the signature of the constructor, and the remaining arguments are any parameters to pass to the constructor.

The signature specified in the fourth parameter is made up of a set of parentheses which contain symbols for a method's parameters. For example, since `JavaAccessor`'s constructor expects no parameters, it simply has the signature "`()`":

```
hJavaAccessor=(HJavaAccessor *)
    execute_java_constructor(0, "JavaAccessor", 0,
        "()");
```

Similarly, `execute_java_dynamic_method` (shown below) also expects at least four parameters. It can require more, depending on the number of parameters the Java method requires. The first parameter indicates the language environment, which again is zero for Java. The second parameter is a reference to the Java object created via the `execute_java_constructor` function. The third parameter is the name of the method to invoke. The fourth is the method's signature, and the remaining arguments are any parameters to pass to the method.

As shown in this example, the Java method "`printRC`" expects a single argument of type `int` (`I`) and does not return any values, so it

Listing 3: Demonstrates Java Object Access

```
public class JavaAccessor
{
   public synchronized native String getEnv(String var)
          throws java.lang.NullPointerException;

   public native int deleteFile(String fileName);
   public int delRC=0;


   public static void main(String args[])
   {
       System.loadLibrary("ja");
       JavaAccessor ja = new JavaAccessor();

       try System.out.println("PATH=" +
                 ja.getEnv("PATH"));
       catch(java.lang.NullPointerException e)
       {
         System.out.println("getEnv returned NULL!");
       }

       ja.deleteFile("dummy.txt");
       System.out.println("Return code from"+
                          " deleteFile() = "+
                          ja.delRC);
   }

   public void printRC(int RC)
   {
       System.out.println("Java: RC=" + RC);
   }
}
// End of file
```

has a V (for void) to indicate this at the right side of the closing parenthesis:

```
if (hJavaAccessor)
    execute_java_dynamic_method(0,
        (HObject *)hJavaAccessor,
        "printRC",
        "(I)V", rc);
```

One good way to find out what letters to use for data types in the signature is to define native methods in a dummy java class and then generate a stub file using the javah utility. You can then inspect the `generated .c file`, which will have comments indicating the signature types. For example, listing 5 shows a stub file generated from the following Java class:

```
class test
{
    public native long LONG();
    public native float FLOAT();
    public native boolean BOOLEAN();
    public native void MIX_PARMS(
        int i, long l,
        String s, byte b,
        float f, double d,
        char c, boolean bool,
        short si, int[] ai);
}
```

Exceptions and Thread Synchronization

C functions called by Java classes can throw exceptions, which can be caught by Java methods. This is accomplished by the `SignalError` C function provided in the Java library. Consider thisline taken from Listing 4:

```
SignalError(0,
    "java/lang/NullPointerException",
    0);
```

Java is multi-threaded language, hence C functions used in Java could potentially be accessed concurrently by various threads in a given Java program. Java provides a data-locking mechanism which permits thread-safe operations in Java and native methods. For C functions written for Java, thread-safe operations can be accomplished by using the `synchronized` keyword in the Java method declaration (as shown below) and three C functions provided in the JDK: `monitorWait`, `monitorNotify`, and `monitorNotifyAll`.

```
public synchronized native String getEnv(String var);
```

Conclusion

Java is a feature-rich language that comes bundled with packages for everything from local file I/O, to socket communications, to GUI programming, to classes for the World Wide Web. Still, there might be times where you need to drop down to C or C++ to accomplish specific tasks, or perhaps to leverage existing C routines or a C API library. For these times, it is nice to know that Java provides the ability to do so. However, one thing to keep

in mind is that, whenever possible, any C functions written for Java should be kept as portable and ANSI compliant as possible. Portability is a big benefit of Java, and it would be nice if you could have the C code be portable as well.

If you do not yet own a book on Java, try downloading the free documentation from Sun Microsystems' FTP site (ftp.javasoft.com). *The Java Tutorial,* by Mary Campione and Kathy Walrath, is an excellent source of information for learning Java. It is available on the Web site in two formats, HTML and Poscript. ❑

Listing 4: *Access to Java environment from C*

```
#include <stdio.h>
#include <string.h>

#include "JavaAccessor.h"


/*** Get environment variable ***/
struct Hjava_lang_String
    *JavaAccessor_getEnv(
        struct HJavaAccessor *javaObj,
        struct Hjava_lang_String *varName)
{
   char *var=makeCString(varName), *value;

   value=getenv(var);
   if (!value)
   {
       SignalError(
           0,
           "java/lang/NullPointerException",
           0);
       return NULL;
   }

   return makeJavaString(value,
                         strlen(value));
}


/*** Delete a local file ***/
long JavaAccessor_deleteFile(
        struct HJavaAccessor *javaObj,
        struct Hjava_lang_String *fileName)
{
   HJavaAccessor *hJavaAccessor;
   int rc=unlink(makeCString(fileName));
   unhand(javaObj)->delRC=rc;

   hJavaAccessor=(HJavaAccessor *)
        execute_java_constructor(
            0,
            "JavaAccessor",
            0,
            "()");

   if (hJavaAccessor)
       execute_java_dynamic_method(
           0,
           (HObject *)hJavaAccessor,
           "printRC",
           "(I)V",
           rc);
   else
      printf(
         "Unable to create hJavaAccessor\n");

   return rc;
}
```

Listing 5: *Stub File generated from Java Class*

```
#include <StubPreamble.h>

/* Stubs for class test */
/* SYMBOL: "test/LONG()J", Java_test_LONG_stub */
stack_item *Java_test_LONG_stub(stack_item *_P_,struct execenv *_EE_) {
        Java8 _tval;
        extern int64_t test_LONG(void *);
        SET_INT64(_tval, _P_, test_LONG(_P_[0].p));
        return _P_ + 2;
}
/* SYMBOL: "test/FLOAT()F", Java_test_FLOAT_stub */
stack_item *Java_test_FLOAT_stub(stack_item *_P_,struct execenv *_EE_) {
        extern float test_FLOAT(void *);
        _P_[0].f = test_FLOAT(_P_[0].p);
        return _P_ + 1;
}
/* SYMBOL: "test/BOOLEAN()Z", Java_test_BOOLEAN_stub */
stack_item *Java_test_BOOLEAN_stub(stack_item *_P_,struct execenv *_EE_) {
        extern long test_BOOLEAN(void *);
        _P_[0].i = (test_BOOLEAN(_P_[0].p) ? TRUE : FALSE);
        return _P_ + 1;
}
/* SYMBOL: "test/MIX_PARMS(IJLjava/lang/String;BFDCZS[I)V", Java_test_MIX_0005fP
ARMS_stub */
stack_item *Java_test_MIX_0005fPARMS_stub(stack_item *_P_,struct execenv *_EE_)
{
        Java8 _t2;
        Java8 _t7;
        extern void test_MIX_PARMS(void *,long,int64_t,void *,long,float,double,
long,long,long,void *);
        (void) test_MIX_PARMS(_P_[0].p,((_P_[1].i)),GET_INT64(_t2, _P_+2),((_P_[
4].p)),((_P_[5].i)),((_P_[6].f)),GET_DOUBLE(_t7, _P_+7),((_P_[9].i)),((_P_[10].i
)),((_P_[11].i)),((_P_[12].p)));
        return _P_;
}

/* End of file */
```

Evan Manning

Floating-point Summation

Addition may be one of the first operations you master when learning mathematics, but it is probably one of the last ones you learn how to do safely on a computer.

Introduction

The summation of large quantities of floating-point numbers is a simple task that arises frequently in practical applications. Some of the most common applications are numerical integration and calculating averages. But this task turns out not to be as straightforward and predictable as it sounds.

First let's look at the obvious way. Writing a routine to sum a series of numbers may seem trivial. Listing 1 is my version of a Simple Summation routine. It uses a local `float` variable `sum` to accumulate a running total of all elements in the array `flt_arr`. This routine is simple, straightforward, and frequently gives bad results.

The problem is that addition of floating-point numbers is different from addition of integers. Integer addition is like doing addition by hand — you line up two numbers one over the other and add one digit at a time, starting from the least significant. Of course computers do this in binary. Floating-point addition includes a step like this too, but first the numbers must be shifted so that corresponding digits line up. But if the two numbers have sufficiently different exponents then some least significant digits will have to be discarded.

For example, if we have a base ten system with one digit of accuracy then valid values include 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 200, and so on. In this system, 10.0 + 1.0 = 10.0 because 11.0 is not a valid value. In such a system, 10.0 would indeed be the best possible answer for the sum of these two numbers, so there is no problem. But if three or more numbers are to be summed then errors accumulate, and the order of the additions becomes important.

If, for another example, there are not enough bits of mantissa to resolve a difference of one part in ten billion, then:

```
(1.0 + 1.0e10) + -1.0e10
  = 1.0e10 + -1.0e10
  = 0.0
```

but

```
1.0 + (1.0e10 + -1.0e10)
  = 1.0 + 0.0
  = 1.0
```

In this example, all the information from one summand was lost because two of the summands were much larger than the other. Usually when you are summing large sets of floating-point numbers there will not be such pronounced differences between individual terms, but errors will grow from the accumulation of many small errors, and from the difference between the running total and the individual terms.

An obvious fix is to add numbers with higher precision. If we start with an array of `float` we should then keep our internal result as a `double`. If we started with an array of `double` we should then keep our result as a `long double`.

Listing 2 is almost like Listing 1 but with the type of sum changed to `double`. This method gives much higher precision than Simple Summation, and is an excellent approach, when available. But if calculations are already being performed in the highest precision available then extending precision is not an option.

Order in the Court

Much of the imprecision stems from the order of operations. Precision is lost most quickly when the magnitude of the two numbers to be added is most different. With summation, one of the operands is always the sum so far, and in general it is much larger than the numbers being added to it. This effect can be minimized by adding the numbers in order from the smallest to the largest. (Smallest in magnitude, that is. A very negative number is still large in the sense that its least significant bit has little precision. Bits will be discarded in adding -0.001 to 1000.0 but not in adding -1000.0 and 1000.0.)

This effect can be minimized by sorting the numbers by magnitude before summing. Listing 3 does this using `qsort` and a comparison function. First it makes a local copy of `flt_arr` in `lcl_arr`, then it sorts this array with `qsort`. Finally, it sums the sorted terms with a running total like the other methods shown so far.

The results are still not good. Sorting is a much slower operation than addition, and the sum still quickly becomes much larger than the numbers being added, so accuracy is not greatly enhanced.

In fact, particular details of the numbers being summed critically impacts how well sorting works. The best case for sorting is where the sum is near zero and the distribution is nearly symmetric. These conditions will keep the sum near the same magnitude as the numbers being added. These conditions are satisfied in test cases 2 and 3 in the test program (Listing 8), but rarely in real life.

Sorting does bring predictability. Sorted Summation always gives the same answer for the same set of numbers, even if those numbers are presented in different orders. This is not entirely true if you use the

Evan Manning has degrees in Applied Physics from Caltech and Stanford. He has been working as a self-taught C programmer in defense and space applications for the past 8 years. Currently he works for Telos Information Systems as a consultant at NASA's Jet Propulsion Laboratory. He can be reached at manning@alumni.caltech.edu.

comparison function in Listing 4. A number and its negation compare equal, and so may show up in any order in the sorted sequence. When you need predictability, use a comparison function like that in Listing 5. This comparison function breaks such ties to attain complete predictability.

Listing 1: *Simple Summation*

```
float f_add(float * flt_arr)
{
  long i;
  float sum = 0.0;

  for (i = 0; i < ARR_SIZE; i++)
    sum += flt_arr[i];
  return sum;
}
/* End of File */
```

Listing 2: *Extended Precision method*

```
float fx_add(float * flt_arr)
{
  long i;
  double sum = 0.0;

  for (i = 0; i < ARR_SIZE; i++)
    sum += flt_arr[i];
  return sum;
}
/* End of File */
```

Another approach to keeping the operands of each addition similar in magnitude is to add in pairs. Listing 6 presents a routine that loops through the array adding pairs of numbers and reducing the size of the array by half each iteration. At each addition, the summands represent the same number of original entries, and so should have similar magnitudes.

Pairwise Summation proves to be both faster and more accurate than Sorted Summation, and further refinements are possible which would improve it further. But it still cannot compete with ...

The Right Way

An even better way to sum many numbers is Kahan Summation [1]. With this method, a correction is maintained along with the current running total, effectively increasing the precision. Listing 7 implements Kahan Summation. Each term is first corrected for the error that has accumulated so far. Then the new sum is calculated by adding this corrected term to the running total. Finally, a new correction term is calculated as the difference between the change in the sums and the corrected term.

Algebraically, this routine seems to do nothing special. Substituting the line:

```
new_sum = sum + corrected_next_term;
```

into:

```
correction
  = (new_sum - sum) -
      corrected_next_term;
```

gives:

```
correction
  = ((sum +
     corrected_next_term) - sum) -
     corrected_next_term;
```

which would simplify to:

```
correction
  = corrected_next_term -
    corrected_next_term
  = 0;
```

for exact numbers. But for real floating-point numbers, the correction term will frequently be nonzero. Using it does improve accuracy. This method is more complicated than the other methods shown here, but that complexity is easily hidden in a function.

Here is an example program. It is organized around a pair of arrays,

```
float flt_array[ARR_SIZE];
double dbl_array[ARR_SIZE];
```

The main routine (Listing 8) runs four tests. For each test the code repeatedly fills these arrays (OUTER_ITER repetitions) and calls the routine `loop` to shuffle and sum the values.

- The first test fills the arrays with uniform random values ranging from 0.0 to 1.0. I use four calls to `rand` for each element.

Listing 3: *Sorted Summation*

```
float fs_add(float * flt_arr)
{
  long i;
  float lcl_arr[ARR_SIZE], sum = 0.0;

  for (i = 0; i < ARR_SIZE; i++)
    lcl_arr[i] = flt_arr[i];
  qsort(lcl_arr, ARR_SIZE, sizeof(float),
        flt_abs_compare);
  for (i = 0; i < ARR_SIZE; i++)
    sum += lcl_arr[i];

  return sum;
}
/* End of File */
```

This assures me of at least 60 bits of randomness even when rand produces only 15 bits (the minimum permitted by the ANSI/ISO standard).
- The second test is like the first except that the arrays are rescaled so that they run from -1.0 to 1.0.
- The third test uses random values with a bell-shaped distribution.
- The final test is a numerical integration to estimate the value of pi by integrating one quarter of a circle.

loop applies all five summation methods on each of the two data sizes (float and double), and then shuffles the arrays and repeats for SHUFFLE_ITERS iterations.

Results

Over a large number of runs a pattern becomes obvious: Simple and Sorted Summation are tied for least accurate, Pairwise Summation is a little better, Kahan Summation is better still, and the champion is Extended Precision, as shown in Listing 2. (I also provide a function, fkx_add, to do Extended Precision Kahan Summation, which is not shown, but is available on the code disk and via ftp. This function is just like the regular precision fk_add, except that the local floating-point variables are defined as doubles, instead of floats. See page 3 for downloading details.)

In my test on one machine I found Simple Summation to be consistently the fastest. Extended Precision was a bit faster than Simple Summation when Simple Summation used float and Extended Precision used double, but was about 50 times slower when Simple Summation used double and Extended Precision used long double. Apparently this platform favors double over float and does not support long double in hardware but simulates it in software.

Sorted Summation was typically about 100 times slower than Simple Summation, rendering it unsuitable for most applications. Because sorting scales as $n*log(n)$ (where n is the number of operands) while all other methods discussed here scale as n, sorting becomes less suitable for larger data sets. Pairwise Summation is about 3-4 times slower than Simple Summation with doubles, and so would be a good bet if it

Listing 4: *Simple comparison*

```
int flt_abs_compare(const void * a,
                    const void * b)
{
  float fa = fabs(*(float *)a);
  float fb = fabs(*(float *)b);
  if (fa == fb)
    return 0;
  else if (fa < fb)
    return -1;
  else
    return 1;
}
/* End of File */
```

Listing 5: *Tie-breaking comparison*

```
int flt_abs_compare(const void * a,
                    const void * b)
{
  float va = *(float *)a;
  float vb = *(float *)b;
  float fa = fabs(va);
  float fb = fabs(vb);
  if (fa < fb)
    return -1;
  else if (fa > fb)
    return 1;
  else if (va < vb)
    return -1;
  else if (va > vb)
    return 1;
  else
    return 0;
}
/* End of File */
```

were not eclipsed by Kahan Summation. Kahan Summation turns out to be about half as fast as Simple Summation when using `double` precision.

Simple Summation, Extended Precision, and Kahan Summation all use very little extra storage space. As implemented here, Sorted Summation uses an additional array the same size as the original and Pairwise Summation uses an array half the original size. But in some applications the summation would be the last operation performed on the array and so the same storage could be reused for the sorted array or the pairwise partial sums. Alternatively, these algorithms could be optimized somewhat to reduce storage if needed.

Listing 6: *Pairwise Summation*

```
float fp_add(float * flt_arr)
{
  long i, j, limit;
  float sum[ARR_SIZE / 2 + 1];

  if (ARR_SIZE == 2)
    return flt_arr[0] + flt_arr[1];
  else if (ARR_SIZE == 1)
    return flt_arr[0];

  for (i = j = 0; i < ARR_SIZE / 2; i++, j += 2)
    sum[i] = flt_arr[j] + flt_arr[j + 1];
  if (ARR_SIZE & 1)
    sum[ARR_SIZE / 2] = flt_arr[ARR_SIZE - 1];
  limit = (ARR_SIZE + 1) / 2;

  while (limit > 2) {
    for (i = j = 0; i < limit / 2; i++, j += 2)
      sum[i] = sum[j] + sum[j + 1];

    if (limit & 1)
      sum[limit / 2] = sum[limit - 1];
    limit = (limit + 1) / 2;
  }

  return sum[0] + sum[1];
}
/* End of File */
```

Mix and Match

Various combinations of the algorithms discussed here may also prove useful. On platforms like mine, where `float` operations are always slower than `double` operations, you should always sum with `double` precision even when the summands are `float`. This gives you higher precision and greater speed. Kahan Summation with all local variables of type `double` would be faster and more accurate than the version of Kahan Summation shown here.

Similarly, on platforms where `long double` operations are not more expensive than `double` operations, any of these methods should be performed with `long double`.

Even though sorting is slow and does not improve accuracy, it does guarantee to give the same sum for different shufflings of a given data set. When you need this predictability, sort first and then perform Kahan Summation or some other method.

Kahan Summation is always a good bet if you don't have specific timing information available for your target platform. If you can do timing on the target platform, then explore the possibility of combining Extended Precision and Kahan Summation.

Final Thoughts

It seemed natural to write the code for this article in C instead of C++ because C++ features might prove a distraction for some readers and because the code is relatively simple. To my surprise, I found the lack of some of my favorite features of C++ jarring. I particularly missed // comments, function templates, reference parameters, and the ability to mix declarations in with my code.

I recommend Reference [1] to all readers; it really does deliver "What Every Computer Scientist Should Know About Floating-Point Arithmetic" (But Was Afraid to Ask). ❑

References

[1] David Goldberg. "What Every Computer Scientist Should Know About Floating-Point Arithmetic," *ACM Computing Surveys*, Vol. 23, #1, March 1991, pp. 5-48.

Listing 7: *Kahan Summation*

```
float fk_add(float * flt_arr)
{
  long i;
  float sum, correction, corrected_next_term, new_sum;

  sum = flt_arr[0];
  correction = 0.0;
  for (i = 1; i < ARR_SIZE; i++)
  {
    corrected_next_term = flt_arr[i] - correction;
    new_sum = sum + corrected_next_term;
    correction = (new_sum - sum) - corrected_next_term;
    sum = new_sum;
  }
  return sum;
}
/* End of File */
```

Listing 8: *Tests performance of summation methods*

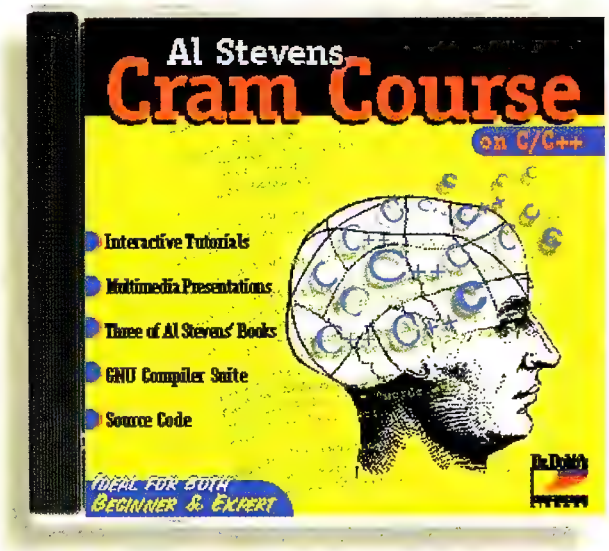
```
int main(void)
{
  float flt_arr[ARR_SIZE];
  double dbl_arr[ARR_SIZE];
  double val;
  double x;
  int j, k;
  long i;


  /* seed random number generator from time */
  srand((unsigned int)time(0));


  /* initialize arrays for test 1: uniform random 0.0 - 1.0 */

  printf("\n%s     Simple      Extended"
         "    Sorted      Pairwise    Kahan\n", "Unif 0-1   ");
  for (j = 0; j < OUTER_ITER; j++)
  {
    for (i = 0; i < ARR_SIZE; i++)
      flt_arr[i] = dbl_arr[i] =
          (  rand()*(double)RAND_MAX*RAND_MAX*(double)RAND_MAX
           + rand()*(double)RAND_MAX*RAND_MAX
           + rand()*(double)RAND_MAX
           + rand())
          /
          (  RAND_MAX*(double)RAND_MAX*RAND_MAX*(double)RAND_MAX
           + RAND_MAX*(double)RAND_MAX*RAND_MAX
           + RAND_MAX*(double)RAND_MAX
           + RAND_MAX);

    loop(flt_arr, dbl_arr, "Unif 0-1   ");
  }
  print_and_reset_minmaxerr();

  /* reinitialize for test 2: uniform random -1.0 - 1.0 */
  printf("\n%s     Simple      Extended"
         "    Sorted      Pairwise    Kahan\n", "Unif -1-1  ");
  for (j = 0; j < OUTER_ITER; j++)
  {
    for (i = 0; i < ARR_SIZE; i++)
      flt_arr[i] = dbl_arr[i] =
          (  rand()*(double)RAND_MAX*RAND_MAX*(double)RAND_MAX
           + rand()*(double)RAND_MAX*RAND_MAX
           + rand()*(double)RAND_MAX
           + rand()) * 2.0
          /
          (  RAND_MAX*(double)RAND_MAX*RAND_MAX*(double)RAND_MAX
           + RAND_MAX*(double)RAND_MAX*RAND_MAX
           + RAND_MAX*(double)RAND_MAX
           + RAND_MAX)
          - 1.0;

    loop(flt_arr, dbl_arr, "Unif -1-1  ");
  }
  print_and_reset_minmaxerr();


  /* reinitialize for test 3: more or less Gaussian */
  /*
   * Simple but not very accurate way to make a gaussian: just
   * sum lots (here 12) uniform random variables.
   */
  printf("\n%s     Simple      Extended"
         "    Sorted      Pairwise    Kahan\n", "Gaussian   ");
  for (j = 0; j < OUTER_ITER; j++)
  {
    for (i = 0; i < ARR_SIZE; i++)
    {
      val = 0.0;
      for (k = 0; k < 12; k++)
        val += rand();
      flt_arr[i] = dbl_arr[i] = val / (double)RAND_MAX - 6.0;
    }

    loop(flt_arr, dbl_arr, "Gaussian   ");
  }
  print_and_reset_minmaxerr();
```

```
  /* reinitialize for test 4:
   * numerical integration to calculate pi */
  /*
   * The area of a circle is pi * radius^2.  When the radius is
   * 1.0 the area is pi.  So we approximate pi as 4 times the
   * area of one quarter of the unit circle, and get the area
   * of that quadrant using numerical integration.
   */
  printf("\n%s     Simple      Extended"
         "    Sorted      Pairwise    Kahan\n", "Pi        ");
  for (j = 0; j < OUTER_ITER; j++)
  {
    for (i = 0; i < ARR_SIZE; i++)
    {
      x = (1 + 2 * i) / (double)(2 * ARR_SIZE);
      flt_arr[i] = dbl_arr[i] =
          sqrt(1.0 - x * x) * (4.0 / ARR_SIZE);
    }

    loop(flt_arr, dbl_arr, "Pi        ");
  }
  print_and_reset_minmaxerr();

  /*
   * Omitted from listing: code to print times to nearest tenth
   * of a second
   * ...
   */

  return EXIT_SUCCESS;
}
/* End of File */
```

Learn C/C++ Programming

THE C/C++ PROGRAMMERS ESSENTIAL RESOURCE

Presenting the
Al Stevens Cram Course on C/C++



Al Stevens, a world renowned programming expert and contributing editor for *Dr Dobb's Journal*, has created the *Al Stevens Cram Course on C/C++* .

This CD-ROM includes the complete text of three books written by Al Stevens, an interactive step-by-step tutorial with precise explanations, video clips of Al discussing important topics, a compiler directly connected to the exercises, lots of usable source code, plus a memory feature that bookmarks your place for quick returns to prior sessions.

Use this CD-ROM to learn C/C++ programming, to answer questions on your programming projects, work through actual examples, compile your example source code, and make sure it's correct right then and there.

No matter what your level of programming expertise, the *Al Stevens Cram Course on C/C++* will help you become a more proficient programmer.

This multimedia CD-ROM features:

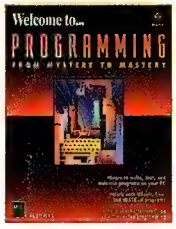
- Three Complete Programming Books
- Interactive Tutorials
- Built in Compiler
- Video Clips
- Bookmark for Quick Returns

See screen shots and a more detailed explanation on our Website: www.ddj.com/cdrom/alstevens.htm

Order Today! 800-548-1971

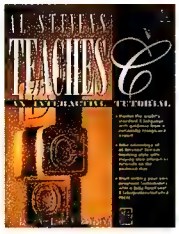
Full Money-Back Guarantee! No Questions Asked!

Includes these Books by Al Stevens:



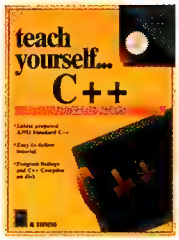
Welcome to Programming

Straight forward explanations of computer files, code, and language makes programming simple.



Al Stevens Teaches C

The perfect introduction to C with an organized, structured approach, teaching good programming through good example.



Teach Yourself ...C++

This book demonstrates the hows and whys of C++ program design in an accessible, informative style.

Dr.Dobb's CD-ROM LIBRARY

SYSTEM REQUIREMENTS:

- 486/40 (486/66 recommended)
- SVGA
- CD-ROM
- Mouse
- 8 MB RAM
- Win95 or NT
- Sound Card (Not essential)
- 5 MB Disk Space

(Will run on Windows 3.1, but requires manual installation)

Complete this form and return to :
Fax: 913-841-2624; E-mail: orders@mfi.com;
Int'l: Use mail, fax, e-mail, or call 913-841-1631

Please send me an *Al Stevens Cram Course on C/C++* CD-ROM today! My price is $69.95 (plus $2.00 U.S./Canada; all other countries $10.00). Please add sales tax in the following states: CA (8.5%); GA (6%); IL (6.25%), KS (6.9%), NY (8.25%), TX (8.25%)

Name

Address

City State ZIP Country

Phone E-mail address

❑ Check ❑ VISA ❑ Mastercard ❑ Amex

Credit Card#: Exp. Date

Signature C0996

Laura Michaels

In Search of a Portable Screen Library

If you thought finding a good screen library was difficult, you're right. But author Michaels supplies some useful hints to aid your search.

This article is for those of you starting a new project and finding yourself in the unenviable position of trying to locate a nice, simple, cross-platform user interface library. I've spent many hours over the past few months in just such a search, checking books, magazines and the Internet. I'm hoping this will save you a great deal of time if you find yourself in a similar position.

Two Extremes

The most basic screen I/O available in C++ is the iostreams library. Personally I'd be happy to just use `cout` and `cin` in all my programs, if only I had not run into some drawbacks. One problem is that not all compilers support `cin` and `cout` correctly, especially if they create programs for Microsoft Windows. Later versions of these compilers may correct this problem, but I must work with what I have. Another major concern is that many users want a more standard user interface, one that looks like the rest of the programs that run under their operating system. Trying to create a user interface that includes menus, edit boxes, push buttons, and the like using just `cout`- and `cin`-style commands would take an enormous amount of work. That's the point of trying to locate the screen library in the first place — to avoid that work.

Going to the other extreme, there are plenty of commercial cross-platform libraries available. But, for a variety of reasons — among them cost and complexity — I quickly ruled them out. Lots of software magazines run some good comparison articles if you think this is the route you want to take.

Having ruled out those two possibilities, I began to explore the options between the two extremes. The biggest difficulty I face in my applications is that I want my user interface library to work on character-based systems (like MS-DOS) as well as graphical systems (like Windows). It's easy to find systems for one or the other category, but finding one that supports both is almost impossible.

Laura Michaels is a senior software engineer at INTERCOMP, creator of entertainment, hobby, and custom software. She can be reached through INTERCOMP, section INT-IP, Box 6514, Delray, FL 33482 or `http://members.aol.com/lauram3017/ \ index.html`.

Character-Based Libraries

Curses

Let's start with character-based user interfaces. One of the most well-known character-based screen libraries is curses (a.k.a. ncurses). If you have programs written in curses or are willing to work with the curses API, you can find several ports of curses available for different systems. The ncurses library works on any POSIX-conformant UNIX platform, and can be obtained by ftp from most GNU collections. (For example, see Item #1 in the box "Starting Places.") A note about downloadable software: whenever retrieving libraries, especially from the Internet, be sure to check that you have the latest stable version. Also, since information on the Internet can change daily, if you can't access a reference, try locating the library or program with a search engine. If you need a DOS port of curses, there's a nice version available from the SIMTEL archive called PDCURSES. (For more information, see Item #2.) It also works on OS/2 and an X11 version is available. There are also ports of curses available for Windows NT and other platforms.

If you find, as I did, that curses doesn't give you enough high-level functionality, you can check for libraries that work on top of curses. A couple of good ones are versions of WXWIN or TK. However, I found out the hard way that just because screen libraries work with curses and you have a DOS version (or a version for whatever system you're on), it does not mean the screen library will work on your system. A lot of libraries that work with curses have other UNIX dependencies in their code as well.

More functionality

I did finally manage in my search to turn up a good number of character-based libraries with higher functionality than curses. They are available for a nominal fee. Several are located at the Simtel archives under the `msdos/cpluspls` directory. There is a library called Sword written for djgpp compiler users. It works with other compilers too. You can check the djgcc collections, such as those under the SIMTEL archive for more information. (See Item #3.)

I also found a very comprehensive library, C/Windows Toolchest, available from Tom Mix software (Item #4). All of these character-based options are reasonably priced. With most of them, you can try before you buy. Another option is Turbo Vision, which Borland freely includes with its C++ compilers. These make a good alternative if you feel very comfortable with the API and you don't need support for other platforms. However, I do need support for other platforms and compilers, so I decided to continue my search.

D-Flat

One last character-based choice I found was the library D-Flat. It is written in C and chronicled in a series of articles in Al Stevens' "C Programming" column of *Dr. Dobb's Journal* [1]. D-Flat is freely available. There are several versions floating around, including a C++ version, an OS-2 version and a djgpp version. However, the latest and most stable version I could turn up was version 20 (Item #5). You can also get a C++ wrapper class (Item #6) to use with version 20. D-Flat 20 works on a DOS

Starting Places

Here is a list of screen libraries mentioned in this article, and places where you can either find them, or more information about them. The Internet changes daily, and we can only supply ftp and web site addresses that were current as of this writing. Most ftp sites have readme files to direct you to new sites if the locations change. Also, you can visit the author's web page at `http://members.aol.com/lauram3017/index.html` for updated information.

1. ncurses	`ftp://prep.ai.mit.edu/pub/gnu/ncurses-1.9.9e.tar.gz`
2. PDCURSES	`http://www.gu.edu.au/gwis/the/markh.html` or `ftp://grind.isca.uiowa.edu/pc/simtel/msdos/screen/ \ pdcurs22.zip`
3. Sword	`http://mimine.iut.univ-metz.fr/~borysgr/sword.web/home.html` `ftp://wuarchive.wustl.edu/systems/msdos/simtel/ \ vendors/djgpp/v1tk/sw200a.txt`
4. C/Windows Toolchest	Tom Mix Software 1132 Commerce Dr., Richardson, TX 75081 +1-214-783-6001. `http://www.mixsoftware.com`
5. D-Flat	`ftp://ftp.mv.com/pub/ddj/packages/dflt20.zip`
6. D-Flat wrappers	`ftp://ftp.mv.com/pub/ddj/packages/dfwrap.zip`
7. wxWindows	`http://www.aiai.ed.ac.uk/~jacs/wxwin.html`
8. YACL	`http://www.cs.sc.edu/~sridhar/yacl.html`
9. TCL/TK	`http://www.sco.com/Technology/tcl/Tcl.html` or `http://www.sunlabs.com:80/research/tcl/`
10. Java	`http://www.javasoft.com/`
11. PIGUI FAQ	`http://www.zeta.org.au/~rosko/pigui.htm`

platform and can be compiled as a 16-bit or 32-bit library for Borland, Microsoft, or Watcom. (Although I could not find specific mention of Symantec support in the latest version, it shouldn't be too hard to get a Symantec version running with support for so many other compilers built in.)

Graphical Libraries

Moving on to the graphical end of the spectrum, I located several comprehensive libraries freely available over the Internet. The first one I found was wxWindows. (See Item #7.) wxWindows supports or has ports in progress for Windows, UNIX, Macintosh, OS/2, Nextstep, VMS, and Amiga. I also found some references to YACL. (See Item #8.) YACL supports Windows and OS/2 platforms and UNIX systems with Motif. One nice thing about YACL: there is a YACL book [2], and it's available in local bookstores. Be aware that YACL has some strings attached if you use it for commercial purposes.

Continuing with freely available libraries, I'd be remiss if I failed to mention TCL/TK. (See Item #9.) TCL often runs as an interpreted language, but the libraries for it are all

written in C. In fact, TCL was orginally designed as a library to be used in C programs. Some libraries also exist to ease the embedding of TCL functions in C or C++ code. TK is the actual screen interface library, but it requires the TCL library to run.

TCL and TK are available on a variety of platforms including Windows (3.1, 95, and NT), UNIX, Macintosh, and OS/2. TK includes both standard GUI versions and a curses version to support character-based applications. The UNIX version is very well supported. Rewrites for Windows and Macintosh are currently in the works and may not be as stable. You can also find a number of recent books about this library in bookstores [3] [4].

Have Some Java

Some would say that Java, with its Abstract Window Toolkit (AWT) is the ultimate cross-platform language. I found some references to the AWT libraries. (See Item #10) The AWT is Java's official screen library. It's similar to C++'s iostreams in that it allows screen I/O across all platforms, but has more functionality in creating GUIs. I personally would love to see something like this for C++, as it would automatically solve my portability problems. The Java syntax is very similar to C/C++, so if you want to be absolutely sure that you're using a portable screen interface, switching to Java to write your code may be a very attractive alternative.

Of course, as with most solutions, this one has drawbacks. Java is available only on specific platforms, such as Windows 95, Windows NT, Solaris 2.x, SPARC, and Macintosh systems. No other platforms are supported by Sun as of this writing. Also, while Java is great for applications that are run over networks, if you're selling an executable, it's not a good choice. At this stage in the language's development, Java is an interpreted compiled language. You compile your code into device independent p-code (using a compiler such as javac) and run it through an interpreter, such as the java program. If you need a single executable program, you'll have to look at other alternatives.

Going Native

A list of screen library options would not be complete without considering native user-interface libraries. If you're on Windows, you can use the Win32, MFC, or OWL APIs. On UNIX, there's Motif and Open Look, etc. Native APIs or frameworks are convenient because they're freely available with the compiler. Problems arise when you have to port to other systems. The good news is that several native APIs have already been ported to various other machines. For example, ports now exist from X-Window calls to Microsoft Windows. If you've already coded your program on one system and don't want to rewrite it, this is one option for porting your code.

Based on what I've read, and from my experience with libraries, if I were going the native route I would probably choose a cross-platform API based on MFC. I would choose MFC mainly because it supports the most platforms. Versions of the MFC API are available for UNIX, Macintosh, and even MS-DOS. MFC is also easier to work with (at least for C++ programmers) than Win32. Beyond these two things, however, I know of no compelling reason for you to abandon a framework or library you're comfortable with.

Now for the drawbacks of native APIs. Many of the ports for other systems run much, much slower than the native version. Also, while the native version is usually freely available along with the compiler, many of the other system ports require you to buy commercial products from various vendors.

If All Else Fails . . .

One last place to look for information and comparisons on portable user interface libraries is in the Platform Independent Graphical User Interface FAQ. (See Item #11). It's also helpful to keep an eye on current developments through magazines and on the Internet to see if anything new eventually surfaces.

What Works for Me

So far, I've run through several options, but none of them have worked perfectly for my situation, and perhaps would not for yours either. As I've mentioned, my biggest problem in locating a screen library is trying to find a user interface that can support both character-based and graphical systems. Many implementors of libraries (both commercial and non-commercial) feel that character-based libraries are on their way out and a waste of their time to implement or support. In answer to that thinking, I would like to quickly point out the cases when a programmer may need a character-based user

interface library, and to remind implementors of a few reasons for using one in the first place.

One reason to prefer a character-based library is that the user may be working with a low-end machine. Believe it or not, some people out there are still running old-style 8086 or 80286 PCs with nothing more than DOS installed. Not every user is a programmer who needs to keep up-to-date with the latest systems. If you're working with limited hardware, you may not have the luxury of using a full-blown graphical operating system either. Some people actually do prefer the look of character-based programs and prefer to purchase them over other programs. It is often easier to read the print on character-based screens than on graphical screens. The moral of this story is: it's important to know who your end user is and be familiar with your market or customer base. Also, it's easier to choose a library that supports more than what you currently need, than to have to search for and port to another new library later on if you find the one you have doesn't support all your goals.

If you're in a position similiar to mine, you're probably wondering if I ever found my ideal screen library. The answer is yes and no. What I finally decided to do was pick some libraries that worked well on the systems I needed to implement them on and create some wrapper classes to make those implementations all appear alike on the different systems. This solution keeps me from writing everything from scratch. For Windows I chose MFC, since code written for it can easily be ported between Windows 3.1, 95, and NT. A port of MFC is also available for the Mac platform. When I have more time, I can change the underlying code to use a native Macintosh API and increase the execution speed of my programs for that platform. For DOS platforms, I decided to use D-Flat and the D-Flat wrapper classes. The controls in this library and the D-Flat wrapper class interface are very similar to the MFC library. This helps decrease coding time for the DOS platform port.

I haven't found any one solution that's perfect for my situation; I still have had to do some coding. I created three main wrapper classes similar to the classes used in MFC. One encapsulates the application, another the frame window, and the third encapsulates the individual MDI (multiple document interface) windows. Any functionality specific to the libraries I'm working with is hidden within these three classes. I then designed a uniform interface for the wrapper classes that lets me write to, get information from, or add controls to a window in the same way regardless of the underlying library I'm using. For example, my MDI Window class wrapper has a text function that lets me add text to the screen the same way on any platform. The member function for sending text to a window appears as follows:

```
void text (char *words,
           position *pos,
           attributes *att);
```

I also took pains to encapsulate data in classes as much as possible. This data includes information such as position of controls or text on screen, color of controls or text, and the actual text being written to or read from the screen. Not only does encapsulating this data help when porting, it also allows for room to grow. The requirements of my screen library and the way information is displayed or the type of information I display may change. By encapsulating the information, it becomes much easier to make these changes later. This also helps minimize the amount of code that will be affected by these changes.

My final screen library design is composed of two parts. I choose a platform-specific screen library for the platform I'm working on. I then encapsulate the screen library in my own classes so that I have a universal subset of commands that will get the job done. I hide the platform-specific code within these classes and use it to create my own screen library. I had to do a lot of searching to find which libraries to use, and to finally come to the conclusion that no one library is right for my specific needs. Hopefully, I've managed to give you enough concrete examples of actual user interface libraries available to make your own search for an ideal screen library easier. ❑

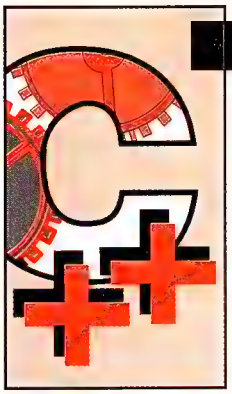
References

[1] Al Stevens. "C Programming" column, *Dr. Dobb's Journal*, May 1991 - May 1995.

[2] M.A. Sridhar. *Building Portable C++ Applications with YACL* (Addison-Wesley, 1996). ISBN:0-201-83276-3.

[3] John K. Ousterhout. *Tcl and the TK Toolkit* (Addison-Wesley, 1994). ISBN: 0-201-63337-X.

[4] Brent B. Welch. *Practical Programming in TCL and TK* (Prentice-Hall PTR, 1995). ISBN: 0-13-182007-9.



Dan Saks

Abstract Declarators, Part 3

Sometimes looking at a problem a different way makes it all come into focus. Dan presents an alternative grammar to make elements of C++ syntax much easier to understand.

A declarator is the central part of an object or function declaration. It includes a *declarator-id* (the name being declared), along with any operators that modify that *declarator-id*. For example, the declarator in

```
unsigned char *const p[N];
```

is `*const p[N]`. It declares that `p` (the *declarator-id*) has type "array with `N` elements of type `const` pointer" to something. The *decl-specifiers* (the stuff to the left of the declarator) tell us that the something is `unsigned char`.

An abstract declarator is a declarator without a *declarator-id*. Abstract declarators appear in numerous C++ constructs, probably most often in parameter declarations and cast expressions. They also appear in exception declarations, exception specifications, template argument lists, and `sizeof` expressions.

For example, a function declaration such as

```
char *memcmp(const void *,
             const void *,
             size_t);
```

has three parameter declarations, yet no parameter names. Therefore, each has an abstract declarator. In the first two parameter declarations, the abstract declarator is just `*`; each parameter's type is "pointer to `const void`". In the third parameter declaration, the abstract declarator is actually empty; the parameter's type is just `size_t`.

Abstract declarators appear in cast expressions as part of the *type-id* (the part of the cast that specifies the result type). For example, in the cast expression `static_cast<D &>(b)`, the `&` is an abstract declarator specifying a reference type. The complete *type-id* `D &` specifies the type "reference to `D`".

For quite a few months now, I've been developing a program called `decl` that parses C++ declarations and translates correct ones into English. At present, `decl` recognizes declarators but not abstract declarators. In my last column (two months ago), I started augmenting the program to recognize abstract declarators, planning to finish it the following month. But, the software business being what it is, things didn't quite work out that way. This month, I'll just say I'll make some more progress.

`decl`'s Current State

`decl` has two main parts: a scanner and a parser. The scanner reads characters from the input stream, filters out the whitespace characters, and partitions the remaining characters into tokens (identifiers, keywords, operators, and punctuation). The parser acquires tokens from the scanner, pieces them into declarations, rejects incorrect declarations, and translates correct ones into English.

Two months ago, I explained how recognizing abstract declarators complicates the parser by occasionally forcing it to look ahead more than one token to determine the next parsing action. (See "C++ Theory and Practice: Abstract Declarators, Part 1," *CUJ*, June 1996.) Last month, I gave the scanner the ability to look ahead and back up by more than one token, in preparation for adding abstract declarators to the parser. (See "C++ Theory and Practice: Abstract Declarators, Part 2," *CUJ*, July 1996.)

It's been several months since I presented the parser in its entirety (see "The Column That Needs a Name: Recovering from Parsing Errors," *CUJ*, April 1996). As I mentioned, that parser recognizes declarators, but not abstract declarators. It also recognizes function declarators, but does not allow parameters in them. This month, I will explain the remaining design issues for a parser that can recognize not only abstract declarators, but also function declarations with non-empty parameter lists.

Left Recursion vs. Iteration

One of the points I've been trying to make with these articles on parsing is that, once you really learn to read it, the C++ grammar can guide you in synthesizing or decomposing any C++ construct. I've been using the `decl` program to show how the grammar maps into a fairly simple method for recognizing C++ programs, called recursive-descent parsing. Many compilers actually use recursive-descent. Understanding the technique will give you greater insight into C++ syntax and the meaning of compiler diagnostics.

Before I add new code to parse abstract declarators and function parameter lists, I want to take another look at the grammar. Table 1 shows a grammar for those C++ declarations that `decl` already recognizes. The grammar lacks some details of complete C++ declarations, but it covers a very useful subset. It also indicates that the parentheses in a *function-suffix* may surround a *parameter-list*. Until now, this *function-suffix* has been just a place-holder; `decl` does not yet recognize function parameters.

As always, I've expressed the grammar in the EBNF notation, summarized in Table 2. This notation is not the same as the notation used in the draft C++ Standard. Rather, the draft uses the notation that K & R (Kernighan and Ritchie [1]) used to describe C.

Dan Saks is the president of Saks & Associates, which offers consulting and training in C++ and C. He is secretary of the ANSI and ISO C++ committees. Dan is coauthor of C++ Programming Guidelines, and codeveloper of the Plum Hall Validation Suite for C++ (both with Thomas Plum). You can reach him at 393 Leander Dr., Springfield OH, 45504-4906, by phone at (513)324-3601 (the area code changes to 937 after September 1996), or electronically at `dsaks@wittenberg.edu`.

One of the problems with the K & R notation is that it has no direct notation for iteration (repetition). Grammars that use this notation must combine recursion with alternation (choices) to simulate iteration. Don't get me wrong; I like recursion, but not to the exclusion of iteration. Using recursion to describe a repetitive construct is not as clear as using iteration. And, as contradictory as this may seem, there are some recursive rules that you can't map directly into a recursive-descent parser. Permit me to explain.

Table 1: *A simplified grammar for C++ object and function declarations*

```
simple-declaration =
    decl-specifier-seq declarator-list .

decl-specifier-seq =
    decl-specifier { decl-specifier } .

decl-specifier =
    type-specifier .

type-specifier =
    cv-qualifier | simple-type-specifier .

simple-type-specifier =
    type-keyword | type-name .

type-name =
    name .

declarator-list =
    declarator { "," declarator } .

declarator =
    direct-declarator |
    ptr-operator declarator .

direct-declarator =
    ( declarator-id | "(" declarator ")" )
        { array-suffix | function-suffix } .

declarator-id =
    identifier .

array-suffix =
    "[" [ constant-expression ] "]" .

constant-expression =
    name | integer-literal .

function-suffix =
    "(" [ parameter-list ] ")" cv-qualifier-seq .

ptr-operator =
    "&" | [ class-name "::" ] "*" cv-qualifier-seq .

class-name =
    name .

cv-qualifier-seq =
    { cv-qualifier } .
```

Table 2: *A summary of EBNF operators*

```
X = Y .  means "an X is defined to be a Y"

X | Y    means "either an X or a Y"

[ X ]    means 0 or 1 occurrence(s) of an X"

{ X }    means "0 or more occurrences of an X"

also

( )      group sub-expressions

" "      enclose a terminal symbol
```

With recursive-descent parsing, each production in the grammar maps (more-or-less) into a separate parsing function. In `decl`, the parser is a class, and each parsing function is a member of that class. For example, the grammar has a production

```
declarator =
    direct-declarator |
    ptr-operator declarator .
```

and so the parser has a member function called `declarator`. Each parsing function returns a string that spells out the meaning of what it just parsed.

The right-hand side of a typical production refers to the names of other productions. Each reference becomes a function call to the corresponding parsing function. For example, the parsing function for *declarator* includes a call to `ptr_operator` followed by a (recursive) call to `declarator`. The | (vertical bar) in a production indicates a choice. In this case, it indicates that a *declarator* can be either a *direct-declarator*, or a *ptr-operator* followed by another *declarator*.

The *declarator* parsing function appears in Listing 1. `input` is the scanner object. `input.current()` returns the current token, and `input.current().kind()` returns the current token's category. The function looks for a token that can begin a *ptr-operator* before calling `ptr_operator`. If it finds no such token, it blindly calls `direct_declarator`. I took a little liberty in translating the grammar into this function. Had I translated the grammar rigorously, the function would check for a token that can begin a *direct-declarator* before calling `direct_declarator`. However, I wrote `declarator` assuming that `direct_declarator` will complain if it is not happy with the token it sees.

The *declarator* parsing function uses recursion, and uses it properly. *ptr-operator*s bind to a declarator from right to left, and recursion is an effective tool for reversing the order of the *ptr-operator*s so they appear from left to right in the output.

For example, the declarator `*&r` means `r` is a reference to a pointer, not a pointer to a reference. Each call to `declarator` that begins with a *ptr-operator* parses that one *ptr-operator* and saves its translation. `declarator` calls itself to parse the remainder of the declarator (which may include more *ptr-operator*s). When the recursive call returns, `declarator` appends the translation of the *ptr-operator* to the end translation of the declarator. Thus, the first-in-last-out nature of recursion reverses the order of the *ptr-operator*s.

Now let's look at the grammar for *direct-declarator*. The draft C++ Standard uses recursion in this rule, too:

```
direct-declarator =
    declarator-id |
    "(" declarator ")" |
    direct-declarator ( array-suffix | function-suffix ) .
```

This rule has three primary alternatives. The first two alternatives are no problem. Each begins with a unique token that leads the parser toward that choice. However, the last alternative is a puzzler. It begins with a reference to *direct-declarator*, which is this very same rule. Thus, the tokens that lead the parser to this alternative are exactly the same as the tokens that lead to the other alternatives. The

Listing 1: *The declarator parsing function*

```
//
// declarator =
//     direct-declarator | ptr-operator declarator .
//
string parser::declarator()
    {
    token::category tc = input.current().kind();
    if (tc == token::AMPERSAND || tc == token::STAR
    || tc == token::NAME)
        {
        string p = ptr_operator();
        return declarator() + p;
        }
    else
        return direct_declarator();
    }
//End of File
```

We thought about getting President Clinton to speak at our D.C. conference.

But we figured you'd rather hear Tim Berners-Lee.

Register for Software Development '96 East
October 28–November 1, Washington, D.C.

As the inventor of the World Wide Web, Tim is considerably more qualified to provide you with insights into web development. As are the 30 other web authorities teaching at SD '96 East.

With 200 classes and 175 tools vendors, SD '96 East is the largest development event on the East Coast. You'll learn concrete skills from expert instructors including Bruce Eckel, Mark Pesce and Richard Hale Shaw. Trade intelligence with the industry's smartest development professionals. And preview the latest tools for intranet development, object-oriented programming, client/server migration and rapid application development.

SOFTWARE DEVELOPMENT

To get your SD '96 East conference catalog, call us at 800.441.8826 or 415.905.2702. Or send us e-mail at sd96east@mfi.com.

We'll provide you with an exceptional opportunity to meet face-to-face with some of the most influential leaders in Washington, D.C.

More tools. More intelligence.

http://www.sd96.com

only way the parser can distinguish the third alternative from the first two is by looking ahead for an *array-suffix* or a *function-suffix*.

Now that the scanner provides unbounded lookahead, looking ahead should be easy, but in this case it isn't. Well, maybe the looking is easy but finding the right token is hard. An *array-suffix* begins with a left bracket and a *function-suffix* begins with a left parenthesis. The parser cannot simply look for the first occurrence of one of these tokens; it must look for the last occurrence that has not been found by a previous lookahead.

For example, consider the declarator `x[10]`. The first time the parser enters `direct_declarator`, it sees `x` as the current token. It must decide if it can simply accept `x` as a *declarator-id*, or call itself and then call `array_suffix` to parse a suffix. Since there is a suffix after `x`, `direct_declarator` does call itself. Each call to `array_suffix` parses exactly one suffix. Therefore, the recursive call to `direct_declarator` must parse just `x`, and leave the `[10]` for `array_suffix`.

Upon entering the recursive call to `direct_declarator`, the current token is still `x`. This call to `direct_declarator` can't tell whether it was called recursively or from another function such as `declarator`. It must decide for itself if it can just accept `x` as the *declarator-id*, or if it must call itself again. There is a suffix after `x`, suggesting that a recursive call is in order. However, that suffix, `[10]`, has already been spoken for (by the previous call to `direct_declarator`), so this recursive call must not parse the `[10]`. Thus, when the parser looks ahead for a suffix, it must stop before reaching the `[10]`.

Parsing a declarator such as `x[10][20]` gets even harder. In general, the parser must somehow exclude one more suffix from the lookahead on each recursive call to `direct_declarator`. This is no simple task.

The production for *direct-declarator* shown above is said to be *left-recursive*, because the recursive reference to *direct-declarator* appears as the leftmost symbol of an alternative on its right hand side. In contrast, the production for `declarator` shown earlier is right-recursive. Right-recursive rules translate pretty easily into recursive-descent parsing functions; left-recursive rules do not. Fortunately, you can often transform a left-recursive production into an iterative one that's much easier to parse. Let's do this for *direct-declarator*.

The left-recursion in *direct-declarator* suggests that the suffixes group from left to right. For example, in `x[10][20]`, `[20]` is the suffix for `x[10]`, and `[10]` is the suffix for `x`. Thus, `x[10][20]` is equivalent to `(x[10])[20]`. A parser can simply read the suffixes from left to right and pass their translations in that order to the output. Therefore, given `x[10][20]`, then `x` is an "array with 10 elements of type array with 20 elements of type" something.

The first two alternatives in the production for *direct-declarator* indicate that a *direct-declarator* need not have any suffix at all. The third alternative indicates that a *direct-declarator* followed by one suffix is still a *direct-declarator*, and thus may be followed by two or more suffixes. The iterative rule for *direct-declarator* is therefore

```
direct-declarator =
    ( declarator-id | "(" declarator ")" )
        { array-suffix | function-suffix } .
```

This is the way it appears in Table 1. The parsing function for this production requires no lookahead at all.

Syntax vs. Semantics

Now let's look at the grammar rules for abstract declarators and parameter lists. A *parameter-list* is a comma-separated list of parameter declarations. The draft C++ Standard describes *parameter-list* using a left-recursive production:

```
parameter-list =
    [ parameter-list "," ] parameter-declaration .
```

Table 3: *A grammar for* parameter-declaration *using left-recursion*

```
parameter-declaration =
    decl-specifier-seq ( declarator | [ abstract-declarator ] ) .

declarator =
    direct-declarator |
    ptr-operator declarator .

direct-declarator =
    declarator-id |
    "(" declarator ")" |
    direct-declarator ( array-suffix | function-suffix ) .

abstract-declarator =
    direct-abstract-declarator |
    ptr-operator [ abstract-declarator ] .

direct-abstract-declarator =
    "(" abstract-declarator ")" |
    [ direct-abstract-declarator ] ( array-suffix | function-suffix ) .
```


Rewriting this using iteration yields a rule that's easier on the eyes as well as the parser:

```
parameter-list =
    parameter-declaration { "," parameter-declaration } .
```

The draft describes a *parameter-declaration* as:

```
parameter-declaration =
    decl-specifier-seq
        ( declarator | [ abstract-declarator ] ) .
```

This is similar to a *simple-declaration*, except that (1) a parameter-declaration can have at most one declarator (rather than a comma-separated list of declarators), and (2) the declarator in a *parameter-declaration* can be abstract.

Remember, an abstract declarator is just like an ordinary declarator, except it doesn't have a *declarator-id*. Thus, the productions for *declarator* and *abstract-declarator* should be nearly identical. Yet the draft C++ Standard opts to keep them as distinct productions, rather than combine them into one. This approach has both advantages and disadvantages.

There are places in the grammar, such as *simple-declaration*, that allow declarators but not abstract declarators. There is also one place, *type-id*, which allows abstract declarators but not ordinary (non-abstract) declarators. If you combine the productions for abstract declarators with declarators, it seems that you lose the ability to distinguish the places which allow only one kind of declarator.

On the other hand, ordinary declarators and abstract declarators are nearly identical, and describing them with different productions obscures their similarities. Table 3 shows the productions for *declarator* and *abstract-declarator* in the left-recursive form used in draft C++ Standard. As expected, the productions for *declarator* and *direct-declarator* are similar to the productions for *abstract-declarator* and *direct-abstract-declarator*, respectively. (The similarities become more apparent if you delete the word *abstract* everywhere.) However, considering that the only difference between a declarator and an *abstract-declarator* is the presence or absence of a *declarator-id*, it's surprising that the productions don't look more alike.

The other problem with using different rules for the different kinds of declarators is that it introduces parsing problems that require elaborate lookahead. Merging *declarator* and *abstract-declarator* into a single set of productions solves most of those problems.

As a first step in merging the productions, I removed the left-recursion and replaced it with iteration. The resulting productions appear in Table 4. I believe these productions make the similarities between *declarators* and *abstract-declarators* more evident. In fact, the only difference (other than the presence of *abstract-* in a bunch of places) is that the *declarator-id* is missing from a *direct-abstract-declarator*.

The productions in Table 4 still pose parsing problems that require looking ahead. A *direct-abstract-declarator* can begin with a left parenthesis, followed by an *abstract-declarator* and a right parenthesis. The parenthesized *abstract-declarator* is optional, so the parser can skip it and go on to look for an *array-suffix* or a *function-suffix*. However, a *function-suffix* also begins with a left parenthesis. Therefore, the parser must look at the next two tokens to distinguish a *function-suffix* from a parenthesized *abstract-declarator*.

For example, consider the abstract declarators `(T &)` and `(T::*)`. The first declarator, `(T &)`, is a *function-suffix*. The second, `(T::*)`, is a parenthesized *abstract-declarator* describing a pointer-to-member. In parsing either of these declarators, the parser enters `direct_abstract_declarator` with the left parenthesis as the current token. If the parser looks at the next token after the parenthesis and sees a type name such as `T`, it still can't tell if it has a parenthesized *abstract-declarator* or a *function-suffix*. It must look at one more token. If that token is not a `::`, then the parenthesis is the start of a *function-suffix*; otherwise, it's grouping around an *abstract-declarator*.

The lookahead problem I just described arises regardless of whether the grammar uses left-recursion or iteration. However, the iterative grammar in Table 4 has one problem that the left-recursive grammar in Table 3 does not. According to Table 4, an *abstract-declarator* can be empty. According to Table 1, the *parameter-list* (between the parentheses) in a *function-suffix* can be omitted. Therefore, a *direct-abstract-declarator* that is just `( )` could be either parentheses around an empty *abstract-declarator*, or a *function-suffix* with the *parameter-list* omitted. This is a *syntactic ambiguity*. That is, there are two different and apparently valid ways to parse the same sequence of tokens.

A careful (very careful) reading of the left-recursive grammar in Table 3 shows that it does not share this problem. According to Table 3, an *abstract-declarator* cannot be empty. Therefore, a *direct-abstract-declarator* that's just `( )` must be a *function-suffix*.

I prefer the grammar in Table 4. It's more readable and easier to map into a parser. The syntactic ambiguity it introduces is most unfortunate. Does this mean we must abandon it? I don't think so. Such ambiguities occur elsewhere in C++, and even in C. Both languages resolve the ambiguities by simply imposing additional semantic rules.

For example, according to C++ grammar, a function declaration such as

```
int f(const T);
```

has either an unnamed parameter of type `const T`, or a named parameter `T` of type `const` (and by default) `int`. The grammar is ambiguous. C++ resolves the conflict according to a semantic rule called the "maximal munch" rule. (See "The C++ Column That Needs a Name: Understanding C++ Declarations," *CUJ*, December 1995.) According to maximal munch, if `T` is a type name in the scope enclosing the function declaration, then the function has an unnamed parameter of type `const T`. Otherwise, it has a named parameter `T` of type `const int`.

The C++ standards committee recently discovered a syntactic ambiguity in the construct `~X()`. According to the grammar, `~X()` means either

- `~(X())`, which applies the `~` operator to a temporary object of type `X` constructed with no arguments, or
- `this->~X()`, which calls the destructor for type `X` applied to `*this`.

The committee opted to add a semantic rule favoring `~(X())`.

I usually prefer an unambiguous grammar to an ambiguous one that resolves ambiguities by semantic rules, but I'm more concerned about the overall utility of the language description. I want the production for *direct-abstract-declarator* in iterative form, so I can combine it with *direct-declarator*, and I can't use the iterative rule for *direct-abstract-declarator* without some rule to specify the meaning of `( )`. That rule is: a *direct-abstract-declarator* consisting of just `( )` is always a *function-suffix*.

Now, at last, I can merge *abstract-declarator* with *declarator* to form a single set of productions. Table 5 shows the resulting simplified productions for *parameter-declaration* and *declarator*.

Merging abstract declarators with declarators introduces a problem with *declarator-list* (used in *simple-declaration*), which can accept declarators but not abstract declarators. The solution to the problem is a couple more semantic rules. Add the following production to the grammar:

Table 4: *A grammar for* parameter-declaration, *where* direct-declarator *and* abstract-direct-declarator *use iteration*

```
parameter-declaration =
    decl-specifier-seq ( declarator | abstract-declarator ) .

declarator =
    direct-declarator |
    ptr-operator declarator .

direct-declarator =
    ( declarator-id | "(" declarator ")" )
        { array-suffix | function-suffix } .

abstract-declarator =
    direct-abstract-declarator |
    ptr-operator abstract-declarator .

direct-abstract-declarator =
    [ "(" abstract-declarator ")" ]
        { array-suffix | function-suffix } .
```

Table 5: *A grammar for parameter-declaration, with declarator and abstract-declarator as one set of productions*

```
parameter-declaration =
    decl-specifier-seq declarator .

declarator =
    direct-declarator |
    ptr-operator declarator .

direct-declarator =
    [ declarator-id | "(" declarator ")" ]
        { array-suffix | function-suffix }
```

```
concrete-declarator =
    declarator.
```

along with a rule explaining that a *concrete-declarator* must have a *declarator-id*. Also add:

```
abstract-declarator =
    declarator.
```

along with a rule explaining that an *abstract-declarator* must not have a *declarator-id*.

Rules such as these already exist in both C++ and C. For example, they both have the production:

```
constant-expression =
    conditional-expression .
```

along with a slew of rules constraining the operands and operators that can appear in a *constant-expression* (so the compiler can evaluate the expression at compile time). The alternative is a lot of extra productions to describe *constant-expression*.

With these productions that distinguish *abstract-declarators* from *concrete-declarators*, the production for *declarator-list* becomes:

```
declarator-list =
    concrete-declarator
        { "," concrete-declarator } .
```

I'll show you the code for the revised parser next time. For those of you who can't wait, you can find it on *CUJ*'s ftp site (see page 3 for ftp information).

Errata

In past articles, I presented versions of the `decl` grammar that defined *decl-specifier-seq* as:

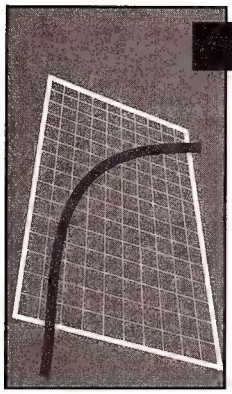
```
decl-specifier-seq =
    { decl-specifier } .
```

This suggests that that the sequence can be empty. It cannot. The grammar in Listing 1 correctly shows that the sequence must contain at least one *decl-specifier*.

Thanks to Steven Lee (`stlee@cisco.com`) for bringing this to my attention. ❑

Reference

[1] Brian Kernighan and Dennis Ritchie. *The C Programming Language* (Prentice Hall, 1978).



Bobby Schmidt

C->C++ Mutations, Part 4

In this installment, Bobby wraps up his series on porting C code to C++. Once again he shows that adapting to C++'s stricter rules is painful, but ultimately results in better code.

What you have in your hot little hands is the fourth and final exploration of C code that mutates behavior when ported to C++.

Type of Character Literals

In the Cretaceous era of Classic C (K&R C), everything seemed to be an `int`. Unspecified function return and object types were `int`s. Pointers were `int`s. `short`s were `int`s (and often still are). `long`s were `int`s. `float`s and `double`s weren't `int`s, but then, real programmers didn't use floating-point [1]. And while `char`s typically weren't `int`s, `char` arguments were promoted to `int`s, and lots of programmers mixed `int` and `char` like beer and pretzels. After all, who among us has never seen or written constructs like [2]

```
char index_to_letter(int const index)
    {
    return 'A' + index;
    }
```

Standard C has maintained much of the `int` legacy. Given this trend, you should be nonplused to learn that so-called "character" literals like `'A'` are actually of type `int`. This leads to the interesting anomaly that, for an ASCII execution character set,

```
char c = 'A';
```

Bobby Schmidt is a freelance writer, teacher, consultant, and programmer. He is also a member of the ANSI/ISO C standards committee, an alumnus of Microsoft, and an original "associate" of (Dan) Saks & Associates. In other career incarnations, Bobby has been a pool hall operator, radio DJ, private investigator, and astronomer. You may summon him at 3543 167th Ct NE #BB-301, Redmond WA 98052; by phone at (206) 881-6990, or via Internet e-mail as `rschmidt@netcom.com`.

is morally equivalent to

```
char c = 65;
```

Even so, I encourage you to use `'A'` here. `'A'` is much more self-documenting, implying that you are thinking of the logical abstracted character `'A'`, and not the physical numerical implementation 65. Also, your code will port correctly to a non-ASCII environment, since `'A'` will map to the appropriate encoding for the new target.

That `'A'` is an `int` is typically invisible to you. The one place you may become aware of it is in statements like

```
if (sizeof('A') == sizeof(int))
    printf(
      "'A' is the size of an int\n");
else if (sizeof('A') == sizeof(char))
    printf(
      "'A' is the size of a char\n");
else
    printf(
      "'A' is of mysterious size\n");
```

This code, built as C, produces

```
'A' is the size of an int
```

The same code built as C++ produces

```
'A' is the size of a char
```

The reason, of course, is that character literals in C++ are truly of type `char`. Why this change from C? To better support function overloading, a concept foreign to C. Consider the two C++ overloaded functions

```
void spew(char const value)
    {
    printf(
      "the value is '%c'\n",
      (char) value);
    }
void spew(int const value)
    {
    printf("the value is %d\n",
          (int) value);
    }
```

If you call `spew` as

```
spew('A');
```

what do you expect to see? Were `'A'` still of type `int`, you'd get the unintuitive result

```
the value is 65
```

Because C++ reckons `'A'` as a `char`, you actually get the more useful

```
the value is 'A'
```

Where Types are Declared

You may think, rationally enough, that all C types must be declared in declarations. But C lets you declare types in expressions also, as demonstrated by the C snippet

```
void *p = (struct s {char c;} *) 0;

struct s x;
```

Here the new type `struct s` is declared in the type-cast expression, allowing us to define objects (like `x`) of that `struct` type. C++ disallows such type-declaring expressions. In C++, all types must be declared in declarations.

One possible C++ rewrite of the above is

```
struct s
    {
    char c;
    };

void *p = 0;
s x;
```

which not only changes where the type is declared, but also takes advantage of the tagged name `s` appearing in the untagged name space.

Vaulting Past Object Initializations

The apparently simple definition

```
purple_doofus Barney = 1;
```

has very different meanings, depending on source language. In C the statement (loosely) means

- Reserve enough storage for a purple_doofus-sized object.
- Name that storage Barney.
- Set the initial value of that storage to 1.

In C++ the definition (again loosely) means

- Reserve enough storage for a purple_doofus-sized object.
- Name that storage Barney.
- Call the purple_doofus constructor that accepts a single integer argument, passing 1 as that argument, and passing &Barney as the constructor's hidden this pointer.

Put another way, initialization in C sets a bit pattern, while initialization in C++ calls a constructor function. That C++ constructor may generate side effects (e.g., allocating resources) requiring collateral side effects (freeing resources) in the destructor. Were an object not properly initialized, use and destruction of that object could produce disaster.

Since C initialization calls no construction function, there are no such side-effect concerns. At worst, an uninitialized C object simply contains the wrong bit pattern, a problem easily correctable by assignment. This laxity allows C compilers to accept code like

```
void f(int const i)
    {
    goto jail;
    if (i)
        {
        int const x = 1;
    jail:
        printf("x = %d\n", (int) x);
        }
    }
```

Because the goto skips x's initialization, you have no clue what will get written by the printf statement.

C++ expressly disallows such leaps past object initialization. Allowing them would induce run-time checks at object destruction, to see if the object's constructor had indeed been called, and would almost certainly lead to unpleasant object behavior.

No Returned Value From Function

Some C functions don't want to return anything useful. In Standard C these functions are prototyped to return void. Cretaceous era C didn't have void, so such functions were always declared to return something. Although their compile-time declarations suggested they had to return a value, at run time these functions didn't actually have to. This prevented programmers from needing to return a dummy value just to satisfy the compiler, when they knew the value would be unexpected and ignored by the caller.

Standard C retains this feature, allowing

```
int initialize(char *p)
    {
    *p = '\0';
    }
```

C++ has had void from the get-go; since there's no compelling reason to write functions like initialize, C++ does not allow this.

Storage Class Specifiers

C supports two storage class specifiers, extern and static [3]. The very phrase "storage class specifier" suggests that these keywords affect things occupying run-time storage. While objects of a particular type take up such storage, the types themselves do not. Nonetheless, according to the C grammar, storage class specifiers are valid (and ignored) in type definitions, giving rise to

```
static struct s
    {
    int i;
    };
```

The C++ grammar prevents such constructs. Among other things, preventing these constructs avoids the confusion that could arise between declaring a class member static, and declaring the entire class static.

Note that, while extern types are disallowed in C++, some C++ implementations hold the notion of exported types. For example, Microsoft Windows DLLs export entities they want to be visible to other applications or DLLs. Traditionally those entities have been objects and functions; however, with the rise of complex class libraries like MFC, DLLs also expose classes, which are conceptually compile-time entities. Exposing a class enables a system to share one copy of the library among all applications, instead of duplicating the class library object code in every client application. Such type exporting is non-standard and definitely non-portable across platforms.

Uninitialized `const` Objects

As bizarre as it may seem, C lets you define uninitialized `const` objects. That is, you can create an object in C with random values that you can never change, as in

```
double const pi; /* ooops, forgot to initialize */

double degrees(double const radians)
    {
    return radians * 180 / pi;
    }
```

C++, with its stronger object integrity, requires you to initialize a `const` object. That C does not require this is quite unfortunate. I don't know why C has this weakness; I'm guessing it's to help accommodate Classic C code ported to Standard C (as Classic C did not have the `const` keyword).

Type of enum

Classic C also didn't have `enum`s. The quintessential workaround was a series of macros, such as

```
typedef int Roy_G_Biv;

#define RED 1
#define ORANGE (RED + 1)
#define YELLOW (ORANGE + 1)
#define GREEN (YELLOW + 1)
#define BLUE (GREEN + 1)
#define INDIGO (BLUE + 1)
#define VIOLET (INDIGO + 1)
```

Ick. As I explored in my January article (first of the Boolean series), macros are evil. Standard C `enum`s are far superior:

```
enum Roy_G_Biv
    {
    RED = 1,
    ORANGE,
    YELLOW,
    GREEN,
    BLUE,
    INDIGO,
    VIOLET
    };
```

Unfortunately, Standard C leaves a gaping type hole in its `enum` support. In C, enumerators (constants like `RED`, defined in an `enum` definition) are of type `int`, meaning enumeration objects can be initialized and assigned from `int`:

```
enum Roy_G_Biv chroma = INDIGO; /* OK */

enum Roy_G_Biv colour = 6;  /* also OK; identical
                               to above example */
```

continued on page 75

C/C++ Users Journal™
Advanced Solutions for C/C++ Programmers

Advertiser Index

You can now get additional information about products and services you see advertised three ways!

❶ Contact the vendor directly using the information in the advertisement. Tell them you saw their ad in *C/C++ Users Journal*.

❷ Fill in the Reader Service number found under the ad or in this index on the bound-in Reader Service Response Card. Provide the requested contact information, then mail or fax (913-841-2624) the card back to us.

cujrs@mfi.com

❸ Email your request to `cujrs@mfi.com`. Make sure you include the magazine's issue number (14.09) and where you would like the information sent.

Advertiser	Reader Service #	Page
KL Group, Inc.	141	48
LEAD Technologies	124	27
LexSaurus Software, Inc.	181	103
Machine Independent Software	140	46
Manpower Technical Services	182	98
MarshallSoft Computing, Inc.	183	100
Mesquite Software	145	53
Micro Digital Inc.	150	59
Micro-Processor Services	184	100
MicroGOLD SOFTWARE	185	102
MicroGOLD SOFTWARE	144	52
MIX Software	156	68
MIX Software	168	87
Mortice Kern Systems, Inc.	105	5
National Engineering Resources	186	103
NobleNet	117	20
Nombas, Inc	123	26
NuMega Technologies	110	11
ObjectSpace	122	25
Omega Point Inc.	111	12
On Time Marketing	143	50
Opt-Tech Data Processing	187	99
Parasoft	119	22
Parsifal Software	188	99
Periphere	189	99
Phar Lap Software Inc.	104	4
Powersoft	114	15
Quadron	130	33
R&D Books	165	82
R-Active	190	103
Raima Corp	208	104
Rainbow Technologies	116	17
Reo Centered Publishing	191	98
Rincon Research Corp	192	101
Ristanovic Case	149	58
Ritz Systems	193	98
Rogue Wave Software	112	13
RSVP Services	194	100
Ryle Design	195	99
Sanders-Indev, Inc	146	53
Sandstone Technology	106	6
Scientific Placement, Inc.	196	101
SD '96 East	**	64
Sequiter Software	101	C3
Sheet Bend Software	197	101
Sigma Software, Inc	198	103
Smoky Mountain Technology	127	30
Snowbound Software	157	70
Softel vdm	**	75
Software Blacksmiths	199	100
Software Blacksmiths	138	45
Software Security, Inc.	120	23
Stingray Software	160	37
Sub Systems, Inc.	158	71
Systems Support Group	115	16
Syware	161	78
Syware	200	98
T&T Computer Products	166	83
Tom Sawyer Software	155	67
Tower Concepts	152	62
Traveltech Staffing, Inc	**	99
Trio Systems	164	81
ULTRA Financial Systems	201	102
UnderWare	118	21
V Communications Inc	109	10
Vireo Software	113	14
Voice Information Systems, Inc	202	99
Volt Computer	203	101
Walnut Creek CDROM	167	85
Web '96 East	**	91
Western Wares	204	103
WIBU-Systems AG	125	27
Willies' Computer Software Co	128	31
Windbase Software	205	99
Wishbone Publishing	206	101
Wonderware	121	24
WROX Press	207	102

This index is provided as a service to our readers. The publisher assumes no liability for errors or omissions.
***This advertiser prefers to be contacted directly.*

The TOTAL Fax imaging SOLUTION

Now there's an easier and faster way to build FAX/Voice/Imaging applications. Black Ice Software provides the tools to build B/W and color applications within a few days. Free of any royalties and multi-platform compatible.

- Fax Development Toolkit
- Print Drivers for Windows, Win 95 or NT
- Color Faxing
- TIFF SDK compression
- Fax C++ FOR WINDOWS, Win95 or NT
- Internet Fax Server
- OCX Custom Controls
- Image SDK PLUS
- Royalty FREE
- GammaLink Fax Boards
- Brooktrout Fax Boards
- Pure Data Fax Boards

Call today for more details!

BLACK ICE SOFTWARE INC

292 Route 101 Amherst, NH 03031

Tel: (603)673-1019 Fax: (603) 672-4112
BBS: (603) 673-6617 Email: blackice@mv.mv.com
CompuServe: 72662,3311

continued from page 71

As I'm sure you've guessed, C++'s type checking does not allow this. Enumerators are of their defining `enum`'s type, meaning `enum` objects can be initialized and assigned only from the same `enum` type. If you are porting C code, you can throw in a type cast like

```
Roy_G_Biv colour = static_cast<Roy_G_Biv>(6);
```

to get around C++'s stronger rules. I find this to be a hack. Enumerations are generally supposed to be abstractions. If you find yourself relying on, or even being aware of, the underlying encoding for an enumerator, you may want to revisit your design.

Another way your ported code may break is if you assume the size of an `enum`. In C,

```
sizeof(INDIGO) == sizeof(int)
```

since enumerators are `int`s. In C++

```
sizeof(INDIGO) == sizeof(Roy_G_Biv)
```

since enumerators are of their defining `enum`'s type. The size of an `enum` is implementation defined, and it may well be that `int`s and `enum`s are the same on your compiler. This is not guaranteed, however, so in general portable code should not make this assumption.

Initializing `char` arrays

This next one helps fix the sort of bugs that drive programmers to become yak herders. Consider my self-aggrandizing program

```
#include <stdio.h>

typedef char const name_part[7];

struct name
    {
    name_part last,
    name_part first;
    };

struct name moi = {"Schmidt", "Robert"};

int main(void)
    {
    printf("CUJ's coolest dude is %s %s!",
            (char *) moi.first,
            (char *) moi.last);
    return 0;
    }
```

This code compiles and links as C, but when run on my Mac gives the less than desirable result

```
CUJ's coolest dude is Robert SchmidtRobert!
```

Who is that, the second cousin of Bond JamesBond? No, it's the result of a subtle bug that snuck by C. C++ is smarter [4], refusing to compile the code as written. If I change the definition of `part_name` to

```
typedef char part_name[8];
```

the code builds cleanly in both languages, yielding the much more reasonable and eminently accurate

```
CUJ's coolest dude is Robert Schmidt!
```

The problem was that the `last` array was too short to hold its initialization value: `last` was seven characters long, while `"Schmidt"` is eight (the seven letters plus the terminating '\0' byte). The letters made it on board, but that final '\0' byte got left at the altar. Because `printf` didn't find the terminating byte in `last`, it sailed past the end of that array, marching right into the immediately following storage (`first`).

C++'s semantics require that a `char` array be defined long enough to contain its initializer. The first version fails that requirement, leading to the compiler diagnostic.

Note that your results for the incorrect C version may differ from mine. For example, a compiler could align such objects on four-byte boundaries, so that up to three extra bytes of padding existed between `last` and `first`.

It's Miller (Freeman) Time!

These past four months, I've highlighted all the big changes, and most of the little ones, that you'll encounter porting valid

Standard C code to C++. Oh, to be sure, there are a few other minor examples; but they are so turgid and dull, you would risk slipping into a coma upon their mention. The full canon of such mutations appears as Appendix C of the Standard C++ Working Paper, and in fact I've based some of my examples on those given in that Appendix. By the time you read this, the committees may have foisted a new version of the Working Paper upon the world; if so, Appendix C could deviate some from what I've covered.

Erratica

As I write, diligent reader Dan Saks is in town for the Seattle leg of his North American rock tour. Over a tasty evening repast at the downtown Bellevue Red Robin, we discussed one of my topics from last month: tagged vs. untagged name spaces. Dan brought up a subtlety which I'd failed to mention, but want to touch on here.

Two of the points I raised in the column are that

- C++ implicitly declares tagged type names in both the tagged and untagged type name spaces.
- The untagged name can be hidden by other declarations for the same name at the same scope, leading to the name being used simultaneously as a type and non-type.

In C, to achieve point 1 above, you must use an explicit `typedef`. This has the side-effect of avoiding point 2. Thus, the C `typedef` serves two distinct purposes: it puts the name in the untagged space, and prevents that name from being trumped by another declaration.

Because C++ gives you point 1 for free, you don't need the `typedef`. If, however, you want to avoid point 2, you must use a `typedef` anyway. Moral: In C++, there are still reasons to use`typedefs`, even though that use may appear redundant.

Dan says colloquially that C++ has one and a half name spaces, since the implicit untagged name space entries are so easily masked. Mister Twister, a.k.a. Marc Briand, *CUJ*'s Managing Editor, suggests this tagged/untagged business would make great fodder, er, subject matter for a future column. I dunno ... I was instead thinking of a series on abstract declarators. Can't get enough of that sexy topic. ❑

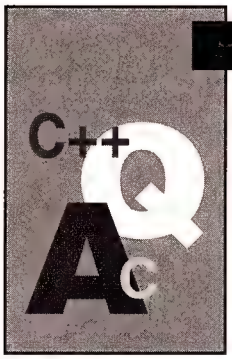
Notes

[1] Not to be confused with the Evergreen Point Floating bridge, the proverbial weak link in the Seattle area's freeway chain.

[2] This function makes gross (but often correct) assumptions about its target character set. For demonstration purposes only. Cars driven by professional drivers on a closed course.

[3] See my December '95 column for an exegesis on these specifiers.

[4] (Sung to the tune of the old calypso song): Let us put C and C++ together and see which one is smarter. Some say C, but I say no, C++ got C like a puppet show. Ain't me, it's WG21 that say C is leading C++ astray. But I say it's C++ today smarter than C in every way ...



Pete Becker

Little-Known Effects of Defining Constructors

Find out why constructors and initializers don't always get along, and when to disobey the gods of OOP.

To ask Pete a question about C or C++, send e-mail to pbecker@wpo.borland.com, *use subject line: Questions and Answers, or write to Pete Becker, C/C++ Users Journal, 1601 W. 23rd St., Ste. 200, Lawrence, KS 66046.*

Q I've been self teaching C++ for about 1 1/2 years now, and I'm comfortable with most of the language features (I mean, it doesn't scare me on a daily basis, as it used to do some months ago). I've tried to compile the following code with two versions of Borland C++ (3.1 and 4.5), and it always give me the error message:

```
Error test.cpp 19: Objects of type
    'C' cannot be initialized with {}

test.cpp:

class A {
public:
     A( char c = '\0' ) : ch( c ) {}
     ~A() {}
     // other members...
private:
     char ch;
};

// this is Ok, no complaint
A vowels[5] = {'a','e','i','o','u'};

struct B {
     char  a, b;
};

// this, obviously, is also Ok
B a_B_struct = { 'a', 'b' };

struct C {
A a, b;
};

// compiler error !!
C a_C_struct = { 'a', 'b' };
```

The compiler accepts the array initialization for both built-in and user-defined types, calling the proper constructor for the last. With structs, however, I get the error message. The help for the message says: "C++ classes can only be initialized with constructors if the class has constructors, private members, functions, or base classes that are virtual." If so, how can the array be initialized with no error?

I've found no reasonable explanation for this "feature," but it seems to be contrary to the (so-called) "spirit of C++," by presenting inconsistent behavior between the language native types and user-defined types.

Is that a language feature a compiler bug? Am I doing anything wrong? Any hints would be greatly appreciated.

— Marcos Capelini

A You're right, the compiler doesn't like your struct A in a context where it would be happy with a built-in type. On the surface this compiler behavior is inconsistent. As Emerson reminds us, however, "A foolish consistency is the hobgoblin of little minds...."[1] So let's look for the reason for this inconsistency, and see if we think it's defensible.

Let's begin by figuring out what the rule here really is. I could start this sort of a discussion off with a citation from the ANSI/ISO working paper, but in reality, I usually do what all of you do: I start off by playing with the compiler to see what it likes and what it doesn't like. Then when I think I've figured out what rules the compiler is applying I go to the authoritative source and try to figure out what rules it says the compiler should be applying. In most cases they're the same.

So, let's start out with something that we know works in C: embedding a built-in type in a struct.

```
Struct s1
{
int i,j;
};

s1 s = { 1, 2 };
```

As I'm sure you expected, this is perfectly acceptable to my compiler. It's straight C code, initializing a static object.

Now let's replace the built-in type with a struct:

```
struct D
{
int a;
};

struct s2
{
D i,j;
};

s2 s = { 1, 2 };
```

This, too, is acceptable, although BC5 gives me two warnings:

```
Warning test.cpp 11: Initialization
    is only partially bracketed
Warning test.cpp 11: Initialization
    is only partially bracketed
```

If you count lines in my source code, you'll see that both warnings are occurring on the very last line, the one that creates our object of type s2 and initializes it. The warning is legitimate: s2 contains two structs, and the initialization can be

Pete Becker is Senior Development Manager for C++ Quality Assurance at Borland International. He has been involved with C++ as a developer and manager at Borland for the past six years, and is Borland's principal representative to the ANSI/ISO C++ standardization committee.

made more specific by adding brackets to delineate the initializers for each of the structs:

```
s2 s = { {1}, {2} };
```

This will quiet the compiler down. The compiler issues this warning because the potential for error is high: when you don't use brackets to set off the initializers for each of the contained structs, you have to count initializers more carefully. Let's change D a bit to see what sort of problems can arise:

```
struct D3
{
int a;
int b;
};

struct s3
{
D3 i,j;
};

s3 s = { 1, 2 };
```

Now the warning is pointing us to a potentially serious problem: the two fields in `s.i` are being initialized, and the two fields in `s.j` are not. That's because in the absence of braces to tell the compiler what to do, the compiler initializes each of the items in the `struct` in the order it encounters them. The leftover fields are left uninitialized. If we really want to do the initialization this way we can leave it alone. If we want to initialize a member of each of the D3 objects, as we did in the previous examples, we need to use a fully bracketed initialization:

```
s3 s = { {1}, {2} };
```

This initializes `s.i.a` and `s.j.a`, and leaves `s.i.b` and `s.j.b` with their default initialization. Finally, if we want to explicitly initialize all the fields in `s3`, we can do so in two ways:

```
s3 sa = { {1,3}, {2,4} };
s3 sb = { 1, 3, 2, 4 };
```

These two initializers do exactly the same thing. It's a bit harder to tell what's happening with the second form, so I prefer the first. Both are valid in C and C++.

So, the problem your code is having is not because you're using a `struct` as a member of `C`. That still works. How is your `struct A` different from my `struct D` above? Well, for one thing, it has a constructor. Let's see if adding a constructor to `D` leads to problems:

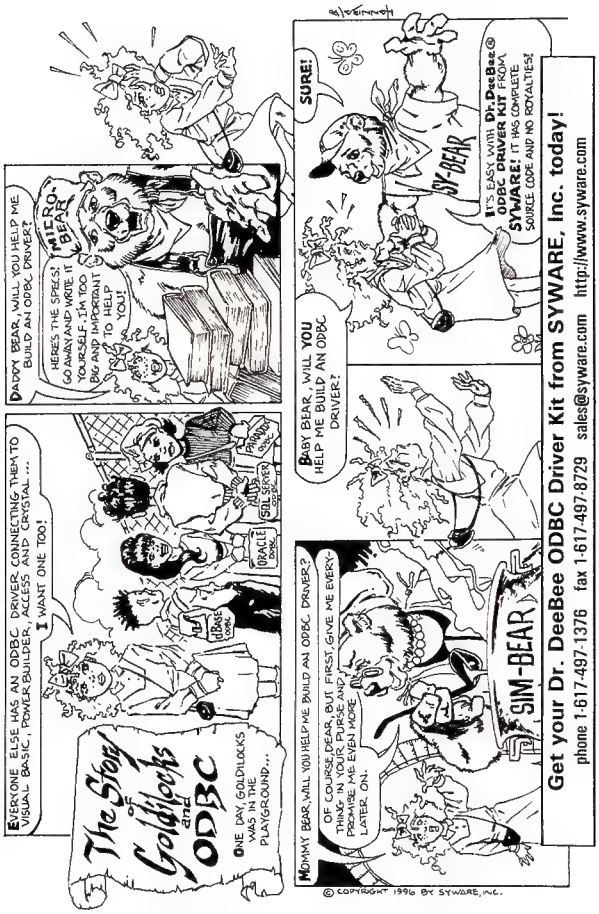
```
struct D4
{
D4(int);
int a;
};

struct s4
{
D4 i,j;
};

s4 s = { 1, 2 };
```

Now I get an error:

```
Error test.cpp 12: Objects of type 's4' cannot be
initialized with { }
```

Does that look familiar? Apparently adding the constructor made the difference. Be careful, though: always dig a little deeper. Let's try a simpler version of this code:

```
struct D4
{
D4(int);
int a;
};

struct s4
{
D4 i,j;
};

s4 s;
```



❑ Request Reader Service #161 ❑

Now I get a different error:

```
Error test.cpp 12: Compiler could not generate default
constructor for class 's4'
```

That's right: we've created a class definition that's completely invalid. We got distracted for a moment by our attempts to apply aggregate initialization, and forgot to look at whether we could create one of these things in the first place. So let's help the compiler out by providing a constructor for our outermost struct:

```
struct D5
{
D5(int);
int a;
};

struct s5
{
s5(int);
D5 i,j;
};

s5 s = { 1, 2 };
```

Now we're back to our original problem:

```
Error test.cpp 13: Objects of type 's5' cannot be
initialized with { }
```

Let's try one more thing: we can get rid of the explicit constructor in s5 if we give D5 a default constructor. Does that work?

```
struct D6
{
D6();
int a;
};

struct s6
{
D6 i,j;
};

s6 s = { 1, 2 };

Warning test.cpp 12: Initialization is only partially
bracketed
Warning test.cpp 12: Initialization is only partially
bracketed
```

Yes, it works. What if we provide a default constructor that does the same thing as the compiler-generated constructor, that is, it simply calls the default constructor for each of the D subobjects?

```
struct D7
{
D7();
int a;
};

struct s7
{
s7() : i(),j() {}
D7 i,j;
};

s7 s = { 1, 2 };
```

```
Error test.cpp 13: Objects of type 's7' cannot be
initialized with { }
```

So it looks like the compiler is happy with aggregate initialization of objects that do not have any user-defined constructors, but won't accept objects that have constructors. We haven't really determined that yet, though: we've only tried default constructors. What happens if we add a constructor that takes an `int`, so it matches the types that we're using in the initializer list?

```
struct D8
{
D8(int);
int a;
};

struct s8
{
s8(int);
D8 i,j;
};

s8 s = { 1, 2 };
```

```
Error test.cpp 13: Objects of type 's8' cannot be
initialized with { }
```

What about a fully bracketed initialization?

```
s8 s = { {1}, {2} };
```

```
Error test.cpp 13: Objects of type 's8' cannot be
initialized with { }
```

Let's try the last resort of the desperate: a cast.

```
s8 s = { s8(1), (2) };
```

```
Error test.cpp 13: Objects of type 's8' cannot be
initialized with { }
```

Still no luck. The rule seems to be ironclad: objects with user-defined constructors cannot be initialized through aggregate initialization. If you look it up in the ARM or in the ANSI/ISO working paper you'll see that that's what it says. You can't do it.

Of course, all this exploration does not get at the answer to the ultimate question: why can't you do it? The logic is quite simple: by adding a constructor to your class you've suggested that the default action that the compiler would have taken would not work correctly. That really shouldn't be much of a surprise: the same thing happens with the default constructor. If you define any constructors in your class the compiler won't generate a default constructor. Add this one to your list of rules: if you define any constructors in your class the compiler won't do aggregate initialization. It believes you when you tell it that it doesn't know enough about your class to do these things right.

[1] "... adored by little statesmen and philosophers and divines." This is from R.W. Emerson, *Essays*, "Self Reliance", (First Series, 1841). I do not, of course, intend to accuse those who seek consistency in programming languages of being in league with "little statesmen."

Q In C++ a class can have public, protected, and private sections. You can give other classes or functions access to the protected and private stuff using `friend`. Is there an acquaintance? That is, if I said that some other class was an acquaintance of my class, the other class could do public (of course) and protected, but not private.

Actually, I'm pretty durn sure that this doesn't exist now in C++. Do you know if it's going to exist in the future? I know that with some finagling I can accomplish what I want to do. I'd rather do it the easy (but not yet existing) way :-) — Brenda Molina

A No, there's no acquaintance, and there's not much chance of getting one. There has been no lack of proposed changes to the Standard's specifications of access specifiers, to provide finer control over who can access what. But for the most part, these proposals tend to be work arounds. That is, they attempt to work around design problems in specific programs, rather than provide broadly useful language changes. In such cases, it's better to fix your program design rather than hack the language definition.

The C++ language definition, whether you get it from the ARM or from the ANSI/ISO working paper, does not tell you how to use the things that it defines. That's something you have to figure out for yourself. Used properly, protected members provide a simple, powerful, and controllable mechanism for permitting changes to the behavior of a class. Make sure you understand how this mechanism works, and that you've used it properly, before you decide that it's too limiting.

Public members of a class `C`, of course, can be accessed by any code that can access an object of type `C`. They define the behavior that the class `C` implements. That is part of the contract between the designer and the user of the class `C`. When `C` has a member function named `show`, and `show` is documented to display the object on which it is called, you ought to be able to rely on that behavior in all objects of type `C` and in all objects of types derived from `C`. If you can't rely on that, someone has messed up the class hierarchy.

Protected members are used to help derived classes implement the documented behavior of the base class. Protected members typically do one of three different things: they provide access to important information about the base class that functions in the derived class may need; they provide implementations of the basic operations that derived classes may need; and they provide hooks so that derived classes can modify the details of how the base class behaves.

Let's take the classic example of a base class for objects that can be drawn on the screen:

```
class Drawable
{
public:
virtual ~Drawable();

void draw() const;
void erase() const;
void move( int x, int y );
protected:
    Drawable( int x, int y );

    virtual void do_erase() const = 0;
    virtual void do_draw() const = 0;

    int get_x_pos() const;
    int get_y_pos() const;
private:
    int x_pos;
    int y_pos;
};
```

```
void Drawable::draw() const
{
do_draw();
}

void Drawable::erase() const
{
do_erase();
}

void Drawable::move( int x, int y )
{
erase();
x_pos = x;
y_pos = y;
draw();
}
```

The documentation for this class should have two parts, one describing the public interface and one describing the protected interface. The public interface is available to users of the class and classes derived from it. This interface supports drawing a derived object, erasing a derived object, moving a derived object, and destroying a derived object. (Of course, the documentation would describe each of these operations in more detail.) The protected interface is available to designers and implementers of classes derived from `Drawable`. The protected interface supports creation of an object at a specified position through the constructor. It also enables those with acess permission to determine the object's current location, through `get_x_pos` and `get_y_pos`. The protected interface also provides hooks through `do_erase` and `do_draw`, so that the implementer of a derived class can implement erasing and drawing appropriately.

Let's look at a `Circle` class based on `Drawable`:

```
class Circle : public Drawable
{
public:
    Circle( int x, int y, int r );
protected:
    int do_draw() const;
    int do_erase() const;
private:
    int radius;
};

Circle::Circle( int x, int y, int r )
    : Drawable(x,y), radius(r)
{
}

int Circle::do_draw() const
{
draw_circle( get_x_pos(), get_y_pos(), radius, black );
}

int Circle::do_erase() const
{
draw_circle( get_x_pos(), get_y_pos(), radius, white );
}
```

The class `Circle` adds its constructor to the public interface inherited from `Drawable`. That is, `Circle` promises us that we can create a circle at a specified location with a specified radius, that we can draw and erase a circle, and that we can move a circle to a different location. To do these things, `Circle` uses protected members of `Drawable` for all three of the basic capabilities that I mentioned earlier. The constructor uses `Drawable`'s constructor to pass the `Circle`'s initial position down to `Drawable`. The functions `do_draw` and `do_erase` use `get_x_pos` and `get_y_pos` to get the basic data they need from `Drawable`, and they provide implementations of `draw` and `erase` that are appropriate for a `Circle`.

Now we can create a `Circle` and move it around:

```
int main()
{
    Drawable *object = new Circle( 0, 0, 10 );
    object->show();
    object->move( 5, 5, );
    object->erase();
    delete object;
    return 0;
}
```

There's a problem with `Circle`, though. Suppose you have an object manager that keeps track of `Drawable` objects, and tries to be smart about not drawing objects that are not currently on the screen. To do that it needs to know the positions of all of the `Drawable`

objects. That seems simple enough: just have the object manager call `get_x_pos` and `get_y_pos` whenever it needs to know where an object is. The problem is, it can't, because these functions are protected. Your first reaction might be to make the `manager` class a `friend` of `Drawable` so that it can call these functions. But this is overkill, since `manager` will then have access to `Drawable`'s private data as well. Wouldn't it be nice if you could say that the object manager can get at `Drawable`'s protected members but not its private members? Alternatively, wouldn't it be nice to let the object manager call `get_x_pos` and `get_y_pos` but no other protected or private members?

Tinkering with access rights might seem like a good answer at first glance, but there's a much better solution: make `get_x_pos` and `get_y_pos` public. After all, you will almost certainly encounter other situations in which you need to find the location of a `Drawable` object. Make this change and the problem goes away, and we have a class with a broader and more usable public interface.

No doubt some of you will object at this point that I've set you up: making `get_x_pos` and `get_y_pos` protected instead of public is obviously wrong; you suspected it all along. I plead guilty, but with an explanation: it's easy to get so focused on the details of making your code work that you lose sight of the overall design. When you run into an access problem, think about your class's public interface and protected interface. Don't just add `friend` declarations or wish for fancier options for controlling access. Look at how you got into your current situation, and consider all the possible ways out. Often the best solution is to rethink the interfaces to your class. If the design is wrong, fix it.

Q We are trying to develop a Windows application that has multi-lingual support. To do this we have implemented a `Msgs` class in C++ that reads in a text file as an array of strings (`TArray<string>`). Since almost every subclass in the application needs to generate error/information messages it seems inefficient to pass the `Msgs` instance down the tree to make its members accessible. Declaring the `Msgs` object as a global would break OO rules (wouldn't it?).

A colleague has tried using a pointer to the parent class containing the `Msgs` object. Since this pointer has a type defined by the parent then the class using the `Msgs` object needs to know that type, thus again breaking the OO rules.

Inheritance could be used but the subclasses are not forms of `Msgs`, but have `Msgs` as part of them. For example:

```
class Msgs
{
private:
    TArray<string> tasMsgs;
public:
    string& operator[](int index);
};

class OtherClass
{
public:
    void error_out(int id)
    {
    cout << tasMsgs[id]; // an error since
                         // OtherClass doesn't
                         // know about tasMsgs
    }
};

class Parent
{
public:
    Msgs mTheMessages;
    OtherClass oAnObject;
};
```

Any suggestions? — Tom Robinson

A There's a saying among those who trek the back country: if the map doesn't match the terrain, trust the terrain. The same thing applies to the "rules" that we use to suggest good design and coding techniques. If the rules don't fit your application, make sure the application works right.

You've mentioned two "rules" of object-oriented programming: global variables are bad and class interdependencies should be minimized. Both are generally good rules; both should also be violated when appropriate, and not taken as absolutes.

Sometimes a program uses data that is truly global. If that's the case, make it global. That's often the case with error messages. Sure, you could define each message in the place where it's needed, but then you run into exactly the problem you're trying to solve:

when you want to localize the application (that is, translate messages into a language appropriate for your customers), it's much easier to have all the messages together in one place. Let's take a look at your code rewritten with the messages stored globally:

```
class Msgs
{
public:
    Msgs();
    string operator[](int index) const;
};

const Msgs ErrorMessages;

class OtherClass
{
public:
    void error_out(int id)
    {
    cout << ErrorMessages[id];
    }
};
```

I've made several changes to the sketch of your class `Msgs`. I added a declaration of a default constructor; I assume this was part of your original class, and you left it out of your note to make the class definition shorter. I added the default constructor back in because it's important to remember that it's there: it's responsible for loading the messages from wherever they're stored, presumably in some file that the program knows about. I also removed the actual array of messages. When we're talking about program design we're not interested in the internal details of how a class does whatever it does. All that matters is that you can index into an object of type `Msgs` and get back a string. I changed the return type of the index operator from `string&` to `string`, and I've added a `const` qualifier to it, to indicate that using it won't modify the object that you're calling it on. That means that the object `ErrorMessages` which I've added can be a `const` object.

Maybe you've noticed my change puts a heavy emphasis on `const`-ness. There's a reason: one of the biggest dangers in using global data is getting confused about which functions change the data. You may find that you no longer know whether the object actually contains what you think it contains. If you forbid your code to modify your global data you can have a great deal more confidence in its integrity. Once the `ErrorMessages` object has been initialized no one is going to change it. Your messages are safe.

Even when making things like `ErrorMessages` const, you can still carry out important optimizations such as lazy evaluation. Suppose you don't want to take the time to read the message file at startup, but instead want to wait until the program needs to display an error message. You can do that and still insist that `ErrorMessages` is a `const` object. We refer to objects like this as *logically const*: calling any member functions on `ErrorMessages` produces the same results, regardless of the preceding sequence of calls to its members. You'll always get the same message when you look up the same index.

To implement this behavior, keep a flag inside `Msgs` to indicate whether you've read the message file, and check that flag in the index operator. Both the flag and the message array will be modified by the index operator on its first pass. Since I've marked the index operator as a `const` operation, we have to tell the compiler that it's okay for `const` member functions to modify these two data members. That's the job of the keyword `mutable`.

```
class Msgs
{
public:
    Msgs() : data_present(false) {}
    string operator[](int) const;
private:
    mutable bool data_present;
    mutable Tarray<string> messages;
};

string Msgs::operator[](int index) const
{
if( data_present == false )
    {
    // insert code to read message file
    data_present = true;
    }
return messages[index];
}
```

The other risk in using global data is that two or more modules may use the same name for different global objects. That's easily solved: just put the global objects into a *namespace*:

```
Namespace Errors
{
class Msgs
{
public:
    Msgs();
    string operator[](int index) const;
};

const Msgs ErrorMessages;
};
```

The construct

```
Namespace Errors
{
. . .
};
```

defines everything between the curly braces to be within the `Errors` namespace scope. To access names defined in this scope, code outside the scope must add the namespace name as a qualifier:

```
class OtherClass
{
public:
    void error_out(int id)
    {
    cout << Errors::ErrorMessages[id]; //qualifier req'd
    }
};
```

So far I've been assuming you have only one set of error messages to be used by all classes in your application. The techniques I've described work fine in that situation. If you want to break your error messages down into smaller groups to make them more manageable or more flexible you've got do to a bit more coding. Let's say you've designed your class `OtherClass` to be used as a member of two separate classes, `Class1` and `Class2`. You want it to display one set of error messages when it is used in `Class1` and a different set when it is used in `Class2`. This situation is fairly common, for example when `OtherClass` is some sort of helper class, providing services to `Class1` and `Class2`. In that case, the program must report a failure in the helper class in terms that make sense for `Class1` and `Class2` respectively. You must be able to tell `OtherClass` which set of messages to use, and that means that you must give it a pointer or reference to a `Msgs` object holding the appropriate messages. Here's a sketch of how this would be done:

```
class Msgs
{
public:
    Msgs(const char *);
    string operator[](int index) const;
};
class OtherClass
{
public:
    OtherClass( const Msgs& msg)
        : ErrorMessages(msg) {};
    void error_out(int id)
    {
    cout << ErrorMessages[id];
    }
private:
    const Msgs& ErrorMessages;
};

class Class1
{
public:
    Class1() : ErrorMessages(
        "Class1msg.txt" );
          other(ErrorMessages) {}
private:
    const Msgs ErrorMessages;
    OtherClass other;
};

class Class2
{
public:
    Class2() : ErrorMessages(
        "Class2msg.txt" );
        other(ErrorMessages) {}
private:
    const Msgs ErrorMessages;
    OtherClass other;
};
```

In every instance of `OtherClass` we've replaced the global variable `ErrorMessages` with a reference to the object that holds its error messages. Now, whenever we create an object of type `OtherClass` we specify the object that it uses to get its error messages. This technique gives us the most flexibility in structuring our error messages. Note that we've done this without requiring `OtherClass` to know anything about the structure of `Class1` or `Class2`. Rather, those classes know their own structure, and know that they need to give an object of type `Msgs` to `OtherClass` for its message handling. `OtherClass` has to know only about `Msgs`, which is information that it already had anyway.

You're right to be concerned about global data and about structural interdependencies. Don't overdo it, though. One of the biggest challenges in program design is to recognize when it's all right to break the rules. ❑

Write for CUJ!

Do you sometimes get the feeling your talents go unnoticed? Do you feel like a cog in the software machine? Put some spark in your career, by writing an article for CUJ. Writing for CUJ is rewarding, pays good money, and gets you recognition for your hard work.

Fire Up the Word Processor

Don't wait for an English degree. You needn't be a professional writer to write for *CUJ*. *C/C++ Users Journal* is written *by* practicing programmers, *for* practicing programmers. If writing is not your strong suit, we have a friendly and experienced editorial staff to help you over the rough spots. Our only requirement is that you have something worthwhile to say. If you've developed a useful technique or tip you want to share with other programmers, we want to hear from you, and so do our other readers.

Write for Real Programmers

CUJ is written for real programmers just like you, who want to see practical treatments of real programming problems, or tutorials that make theory or complex issues more accessible. Your article should concern C or C++, and should provide examples of working code whereever possible. We also publish Book Reviews, User Reports, and very short articles (< 1000 words).

Solid Beats Pretty

When it comes to code, we're picky about the things that count. We're not fussy about where you put your curly braces; we do care about good design and well-crafted code. While we recognize that occasional bugs are inevitable, we expect would-be authors to make a diligent effort to eliminate them before submitting an article.

How to Submit an Article

If you have an idea for an article, send us a proposal: a brief abstract and outline, and a brief resume of your qualifications. Ask for a copy of our Author Guidelines for guidance in writing a manuscript. You must submit your manuscript in electronically readable form. Files must be in raw ASCII format, flush left, with one empty line between paragraphs. Submit proposals, manuscripts, and Author Guideline requests to:

Marc Briand
C/C++ Users Journal
1601 W. 23rd St., Suite 200
Lawrence, KS 66046-2700

PH: (913)-841-1631
FAX:(913)-841-2624
e-mail: marc@rdpub.com
mbriand@mfi.com

Each month we print both theme and non-theme articles. Any time is a good time to submit an article to *CUJ*, but to print it as a theme article we must receive it by the deadlines shown on our Editorial Calendar. Check out our Editorial Calendar on this page if you're looking for article ideas.

1996 & 1997 Editorial Calendar

Issue	Theme	Drafts Due
December	Optimization and Efficiency	16 Aug 1996
January '97	Cross-Platform Development	13 Sept 1996
February '97	Real-Time/Embedded Systems	11 Oct 1996
March '97	UNIX	8 Nov 1996
April '97	Database/Persistent Objects	6 Dec 1996
May '97	Internet Development	3 Jan 1997

Upcoming Themes

Optimization and Efficiency. However powerful computers become, efficiency is always an important issue. Serious programmers need tools for determining where a program spends its time, and reports that highlight how a program consumes storage. They also need techniques for making programs faster and smaller, with a minimum loss of readability and maintainability. We are looking for articles on how better to measure C and C++ programs, and how to improve their performance. We strongly prefer *quantitative* measures to qualitative claims of improvement.

Cross-Platform Development. We are looking for articles that solve tough problems in cross-platform development, and deal with topics such as the following: C++ classes that encapsulate system-dependent interfaces; improving portability of floating-point code; small libraries that provide useful services across many implementations; tools that check for nonportable constructs; judicious use of Java with C/C++ for platform independence; comparing STL bugs and capabilities across platforms; creating cross-platform persistent objects; platform capability test suites.
Overworked topics (require fresh perspective): Little vs. Big Endian problems.

Real-Time/Embedded Systems. Possible topics include: how to adapt (pervert) popular hosted C/C++ compilers to produce embedded code; testing embedded software on conventional systems; networking embedded systems; using Unix/DOS/Windows for data acquisition and control; C++ on embedded systems; performance prediction, estimation, and/or measurement on real-time systems; failsafe mechanisms for mission-critical code; controlling peripheral devices from C/C++ programs.

Special Request

Templates are standard in C++, but today's compilers all implement them differently. We'd like to see a comparison of template handling abilities in the current crop of popular compilers — differences in syntax, instantiaton, what you can and can't do, and implications of the above.

Other Topics of Interest

Java
Memory Management
X-Windows Exceptions
Multithreading Algorithms
Tips 'N' Tricks
OOP

Victor R. Volkman

STL Help on the Web

You don't have to be a C++ expert to use STL. The World Wide Web is a great place to find tools, tutorials, and moral support.

Focus on STL

The WWW has historically been one of the most informative sources of information about leading-edge programming technologies. Though STL has been around for a few years now, it has only recently made its way out of computer science labs and into the toolkits of corporate application developers. My quick survey of WWW via the AltaVista search engine yielded more than 1,000 documents. After considerable weeding, I'm happy to share with you some of the more relevant pages in several subcategories: tutorials, vendor-specific pages, and pointer-pages:

STL Tutorials

Jak Kirman <jak@cs.brown.edu> of the CS faculty at Brown University has thrown together what he considers "a very modest tutorial" on STL. You can read it as a hypertext document or print it out in Postscript for offline perusal. Kirman motivates the discussion with a look at the STL `sort` template while introducing each of the five categories of STL components one at a time. Other sections address the philosophy of STL and suggest how to extend STL to meet your own needs. Kirman makes good use of graphical figures to illustrate his points. Check it out for yourself at:

```
http://www.cs.brown.edu/people/ \
jak/programming/stl-tutorial/ \
tutorial.html
```

Victor R. Volkman received a BS in Computer Science from Michigan Technological University. He has been a frequent contributor to the *C/C++ Users Journal* since 1987. He is the author of the book *Windows Programming with Shareware Tools*. He can be reached at the HAL 9000 BBS (313)663-4173, URL `http://www.HAL9K.com/home.htm`, or email to `sysop@hal9k.com`.

Mumit Khan <khan@xraylith.wisc.edu> at the Center for X-Ray Lithography at the University of Wisconsin provides an exhaustive amount of advice and coding examples (163Kb) in his "STL Newbie Guide." Major headings include: Writing Container Objects, Pointers and STL, STL Iterators, Persistent STL, and a new section about Predicates, Comparators, and General Functions. In addition to how-to information, he specifically addresses common pitfalls, such as "Gotchas with Storing Pointers to Objects." Its all at:

```
http://www.xraylith.wisc.edu/ \
~khan/software/stl/           \
STL.newbie.html#class_howto
```

David R. Musser <musser@cs.rpi.edu> has collected some introductory information, example applications that use STL, and pointers to other STL reference sources. Specifically, Musser's "Standard Template Library" page includes example source code for demonstrating STL vectors and vector iterators, an anagram checking application, and searching and sorting user-defined records. More advanced users may want to look at the official ANSI/ISO STL specifications (available in Postscript form) there.

```
http://www.cs.rpi.edu/~musser/ \
stl.html
```

David R. Musser and Atul Saini, authors of *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library* [see a review of this book elsewhere in this issue — mb] offer a set of source code resources specific to their book. More than 70 example programs are available and you can browse selected parts of the book. Certainly, the website will help you decide whether Musser and Saini's book would be appropriate for you.

```
http://www.aw.com/cp/ \
musser-saini-source.html
```

STL Vendor Specifics

G. Bowden Wise <wiseb@cs.rpi.edu>, of the CS faculty at Rensselaer Polytechnic Institute, has published a pragmatic set of notes entitled "Using the The Standard Template Library with Borland C++." In this document, he describes how to make STL work properly with Borland C++ 4.5 and optionally with Borland's Object Windows Library (OWL). Specifically, he details required changes to `algobase.h`, `funtion.h`, `defalloc.h` and other header files. Also appearing are detailed explanations of the known warnings that the Borland compiler emits while processing STL. You can read about it at:

```
http://www.cs.rpi.edu/~wiseb/ \
stl-borland.html
```

or download Wise's patched STL files directly at:

```
http://www.cs.rpi.edu/~wiseb/ \
stl/stlexp.zip
```

Microsoft's "MFC and Standard Template Library (STL) Alert" page provides a blistering disclaimer in which it places sole responsibility for STL back on Hewlett-Packard (who originally made STL freely available, but retained the copyright). Furthermore, it goes on to say that Microsoft Product Support Services will not assist in technical support issues on STL. Nevertheless, it does highlight required changes to header files (involving namespaces) that you can make yourself to achieve compatability with the Windows SDK or Microsoft Foundation Classes (MFC) and STL. Visit them at:

```
http://www.microsoft.com/     \
visualc/v4/v4tech/stlchg.htm
```

Warren Young <tangent@cyberport.com> publishes the "STL Resource List," an extensive online catalog of pointers to pages detailing the STL support of C++ compilers. It also contains pointers to FTP sites where you can download a specially-tweaked version of the STL files for your specific compiler. Among the STL vendors cited are:

- Symantec C++ 7.x for Windows and 8.x for Macintosh
- Metrowerks C++ 7.0 (for Macintosh)
- Watcom C++ 10.0
- GNU C++ 2.7.1 and 2.7.2
- Rogue Wave's tools.h++ (enhanced version of STL)
- ObjectSpace's STL <Toolkit> (a multi-platform STL)

```
http://www.cyberport.com/~tangent/programming/    \
stlres.html
```

STL Pointer Pages

Here's some "pointers to pointers." These pages provide a rich set of jumping-off points for your own exploration of STL:

```
http://www.cs.rpi.edu/~wiseb/stl-notes.html

http://www.xraylith.wisc.edu/~khan/software/stl/  \
STL.newbie.html#resources

http://cs.ucr.edu/~jcharvat/stl.html
```

STL Usenet Newsgroup Resources

Usenet newsgroups provide an ongoing discussion about the pros and cons of STL implementations. Jan Charvat, a graduate student in Computer Science at the University of California (Riverside), recommends the following newsgroups for STL coverage:

- comp.std.c++
- comp.lang.c++
- comp.lang.c++.moderated

Before posting to Usenet, make sure you've read the STL FAQ, available at:

```
ftp://butler.hpl.hp.com/stl/stl.faq
```

Reader Suggested Links

Paulo Eduardo Neves <neves@lmf-di.puc-rio.br> at Laboratorio de Metodos Formais in Rio de Janeiro, Brazil submits his favorite set of C/C++ links. As usual, I'll provide a brief description of each.

The Code Economics project (CodEc) provides a useful set of links to freeware linear algebra (matrices) and numerical analysis tools for C/C++. It also hosts the source and documentation for Chris Burchenhall's MatClass and Robert Davies NewMat08. The MatClass C++ matrix class library includes makefiles for Borland and Microsoft C++. The NewMat08 matrix class library supports

Borland, Microsoft, AT&T, Zortech, Watcom, GNU, and most other C++ platforms.

http://netec.mcc.ac.uk/~adnetec/CodEc/C_Cpp.html

Custom Innovative Solutions Corporation publishes its C source code in shareware form on the web. You're free to look at and download the source code. If you find the code useful in a commercial software setting, you are asked to pay a small licensing fee. Specific categories include: CGI Tools, File Handling Utilities, Graphics, Indexing and Searching, Numeric Conversions, Parameters and Configuration, Patterns and Parsing. I was most impressed by CNVL2.C, which can convert a long integer to a printable representation in any numbering base (from base 2 to base 90).

http://www.cisc.com/src/c/main.html

The online hypertext version of the GNU C Library reference manual is very convenient to use. You have instant access at your fingertips using the three-tiered table of contents to a variety of topics. Here's just a sampling of the topics:

- Pipes and FIFOs
- Searching and sorting
- Pattern matching and shell-style word expansion
- Calendars, timezones, and alarms
- Defining and using signal handlers
- Locales, internationalization, and extended character sets

http://www.ia.pw.edu.pl/tex-info/libc/libc_toc.html#SEC492

Ajay Shah <ajayshah@cmie.ernet.in> at the Centre for Monitoring Indian Economy, Bombay, compiles the "Free C/C++ Sources for Numerical Computation" page. As the name implies, it is an index of pointers to other types of information, including:

- Free source code available on the net
- Books which come with source code, and hence act as low-cost libraries
- Articles and documents, especially those available over the net

Unlike many other resource lists, this one specifically cites the author name, supported platforms, FTP site, and version information.

http://gauge.phys.uva.nl:2001/c-sources.html

or try

ftp://ftp.usc.edu:/pub/C-numanal/numcomp-free-c.gz

The "C++ Virtual Library," maintained by Lutz Lilje <lilje@desy.de> is a website that has something for everyone from the C++ newbie to those on the bleeding edge of the C++ standards. As you might expect, such a site is primarily focused on providing useful links to authoritative sources of information. Lilje's

page is one of the most active programming resources on the web and has earned the coveted "Four-Star Site" award from Magellan for his efforts. Specific areas of interest include:

- Getting Started: Documents and sources on C++ and OOP
- Learning C++: Virtual courses and tutorials
- Free Packages: Freely available C++ packages of all kinds
- Conferences: List of OOP and Computing conferences
- OOLP: Discussion on Object-Oriented Literate Programming
- Tools & Products: Descriptions of products
- General OO: Object-Oriented programming resources.

There are few sites I visit often enough to merit a bookmark, but the "C++ Virtual Library" is definitely on my hot links list.

`http://info.desy.de/user/projects/C++.html`

If you would like to share some of your favorite C/C++ sources, please email me at `sysop@HAL9K.com`.

C/C++ Mailbag

Here's a sampling of queries which have come to me recently via e-mail.

`Larry_Siden@msn.com` writes:

At my work, we might be able to use a program that can analyze C source code and generate a call tree along with various programming metrics. Do you know of any shareware like that? There's commercial stuff, but it costs in the thousands. I'd like to find something more reasonably priced.

C/C++ Answer Man:

There's no need to pay a fortune for good source code analysis (how about just $47 for a great shareware program?). Try C Exploration tools for Windows. Its C Function Tree and C Structure Tree make program and data structure logic as clear as can be. As for metrics, I'll highlight only a few of them here:

- *Number of source code lines for every file*
- *Number of included files for every source file*
- *For every defined function:*
  - *Number of lines*
  - *Number of functions called*
  - *Number of flow control statements*
  - *Maximum brace nesting level*
  - *Whether it's used only in this file*
- *Function or data type reference list for every file*
- *Location of multiply defined functions and data types*
- *Location of all overloaded C++ functions*
- *`#include` file dependencies for every source file*
- *`#include` file tree for nested `#includes`*
- *Cross-reference for every function or data type*
- *Parent/child relationship for functions & data types*
- *C++ class inheritance tree*
- *C structure/union byte offset calculation*

Check out volume #437 on the CUG CD-ROM or by FTP at

`ftp://ftp.cdrom.com/pub/cica/win3/programr/cxtw107.zip`

`75300.2257@compuserve.com` writes:

I would prefer a better method of searching for information and code on the CUG CD-ROM (please bear in mind that my only exposure is the 1994 CD). A hyper-text style index with search functions is preferable.

C/C++ Answer Man:

Ask no more! The June 1996 edition of C/C++ Users Group CD-ROM includes a hypertext index of volumes 400-437, by Subject keywords, Title, Language, and Platform (OS/CPU). Whats more, you even get a free *licensed version of the popular I-VIEW offline HTML viewer (*`http://www.talentcom.com/iview/iview.htm`*) to use with it. A hypertext index of CUG CD-ROM is available online at* `http://www.HAL9K.com/cug/`.

`d8342012@ccunix.ccu.edu.tw` writes:

I am interested in simulation, especially discrete event-driven simulations.

C/C++ Answer Man:

Well, since you didn't specifically ask for C or C++, I'll assume that C++ class libraries will work for you. I have two specific packages I think you should look at: C++SIM and CNCL.

The C++SIM discrete event process based simulation package provides Simula-style class libraries. C++SIM was written by M.C. Little and D. McCue at the Department of Computing Science in the University of Newcastle upon Tyne (England). The same distribution also includes the SIMSET linked list manipulation facilities. According to MacLennan (1983), Simula was the first computer language to incorporate the ideas of "class" and "object" constructs in 1967. C++SIM currently claims usability only on UNIX workstations, such as SUN Sparcs. C++SIM version 1.0 (released 06/15/92) is now available as volume #394 on the CUG CD-ROM. See also the C++SIM home page:

`http://ulgham.ncl.ac.uk/C++SIM/homepage.html`

The Communication Networks Class Library (CNCL) is a C++ library created by the Communication Networks department of Aachen, University of Technology, Germany. CNCL is both a class library featuring generic C++ classes as well as a simulation library with strong points in random number generation as well as statistical and event-driven simulation. CNCL v1.4 (released 01/05/96) is available on the CUG CD-ROM as volume #443. See also the CNCL home page:

`http://www.comnets.rwth-aachen.de/doc/cncl.html`

❑

New Products

Industry-Related News & Announcements

ImageFX Introduces FXTools 4.0 Professional Edition

ImageFX has released FXTools 4.0 Professional Edition, a royalty-free toolkit, which adds imaging and multimedia special effects technology to Visual Basic, Visual C++, Delphi 2.0, and other development environments that can host 32-bit ActiveX or 16-bit VBX controls in Windows 3.x, Windows 95, or Windows NT.

FXTools 4.0 contains eight reusable ActiveX and VBX components that give developers control over the display of images, text, and shapes, along with easy playback of sound and video. Each control was designed specifically for interactive development. The sound control supports MCI wave and MIDI audio along with 32-channel sound mixing using DirectSound. The video control supports AVI, MOV, and MPEG file formats. Individual video frames can be displayed with special effects.

FXTools 4.0 costs $399. Demos can be downloaded from the Internet (http://www.imagefx.com) or from CompuServe (GO IMAGEFX).

For more information contact ImageFX, 3021 Brighton-Henrietta TL Rd., Rochester, NY 14623-2749. +1-716-272-8030. FAX: +1-716-272-1873. e-mail: imagefx@imagefx.com. WWW: http://www.imagefx.com.

Antares Announces ObjectStar 3.0

Antares Alliance Group has announced the release of Huron ObjectStar 3.0, a cross-platform application development environment for use in the migration of mainframe and desktop legacy applications to three-tiered client/server systems. ObjectStar enables developers to build and distribute mission-critical applications selectively, eliminating the risks of mass migration of legacy applications.

The three-tiered model distributes data, application logic, and user interface independently. ObjectStar provides transaction processing monitors, object brokers, and other software needed to implement three-tiered computing for high transaction volumes and for applications supporting thousands of users.

ObjectStar 3.0 starts at $8,000 per developer seat.

For more information contact Antares Alliance Group, 17304 Preston Rd., Ste. 1200, Dallas, TX 75252. +1-214-447-5500. FAX:+1-214-447-5783.

IPT Releases SNIP for Windows

IPT Corporation has introduced SNIP, a C++ programming tool that allows users to specify their code patterns, describe object model design, and generate C++ classes for those design models. SNIP users enter their Data Structure Model and object relationships using the standardized rules for their corporation, group, or team. SNIP can be extended with a company's own code generation templates to capture entire coding strategies that make sense in a company's application area. Code results will be obtained automatically with SNIP, because patterns are applied to the code creation process.

SNIP is available for Windows NT, Windows 95, and as a command-line version for UNIX.

For more information contact IPT Corporation, 1076 East Meadow Circle, Palo Alto, CA 94303. 800-944-5468 or +1-415-494-7500. FAX: +1-415-494-2758. e-mail: sales@iptcorp.com. WWW: http://www.iptweb.com/

Thomson Announces Talkinx for Inter-Process Communication

Thomson Software Products has announced the release of Talkinx, a development environment for building applications that require inter-process communication such as client/server, peer-to-peer, and three-tier. Talkinx runs on multiple platforms including UNIX and Windows.

Talkinx supports distributed three-tier applications with a six-function API. The included object library links to the application at build time.

Talkinx generates GUI events asynchronously, eliminating polling by applications. Talkinx also supports other native GUI builders for X and Motif.

Priced at $2,995, Talkinx is currently available on most UNIX platforms including Sun Solaris, SunOS, HP/UX, SGI Irix, IBM AIX, and OSF/1. Talkinx ships without runtime royalties.

For more information contact Thomson Software Products, 10251 Vista Sorrento Pkwy., Ste. 300, San Diego, CA 92121. 1-619-457-2700. FAX: +1-619-452-2117.

Matra Datavision Announces CAS.CADE v1.3

Matra Datavision has announced the release of CAS.CADE v1.3, an object-oriented software application development platform that provides programmers with a series of reusable software components and a set of predefined object libraries. CAS.CADE includes a class browser that allows the developer to go directly into the data dictionary and navigate graphically to retrieve documentation about the reusable software components. Using cut-and-paste functions, the developer can insert these components

SQA Announces SQA Suite 5 for Windows 95 and NT

SQA, Inc. has announced SQA Suite 5, a testing solution for Windows NT and Windows 95 that includes a scaleable, client/server test repository. The SQA Test Repository integrates the entire testing process and all users of any SQA product on a local or wide area network. Implementing the SQA Test Repository on a large client/server database allows users to implement an enterprise-level client/server testing project. Based on the Sybase SQL Anywhere scaleable database, the SQA Test Repository client/server database offers the portability and recovery architecture of SQL. In addition, SQA offers a repository implementation option for single-user or small workgroup implementations, which uses the Microsoft Access database.

SQA Suite 5 comes in two versions. SQA Suite: TeamTest Edition, for $2,995 per user, includes SQA Robot and SQA Manager. SQA Suite: Client/Server Edition, starting at $12,395, includes SQA Robot, SQA Manager, and SQA LoadTest.

For more information contact SQA, Inc., 10 State St., Woburn, MA 01801. +1-617-932-0110. FAX: +1-617-932-3280. WWW: http://www.sqa.com.

where they are needed in the application program. In the CAS.CADE development language, improved memory management reduces the size of code generated to improve engine performance.

For more information contact Matra Datavision, Inc., One Tech Dr., Andover, MA 01810. WWW:http://www.mdtvus.com.

Micro Digital Releases smx v3.2

Micro Digital has released their smx v3.2 real-time kernel. smx's main modules are the smx kernel, DOS-compatible file manager, TCP/IP stack, dynamic load module system, smx++ class library, task debugger, protected-mode environment, and windowing utility. These are still sold separately and may be purchased in any combination. One new feature of version 3.2 is an integrated platform that assures any combination of modules will integrate smoothly together.

smx v3.2 supports real mode, 16-bit segmented-protected mode, and 32-bit flat-protected mode. All modules are supported in all three modes. smx also supports the latest versions of 16- and 32-bit Visual C++, Borland C++, Paradigm DEBUG and LOCATE, Soft-Scope, CSiLocate, and Periscope.

smx is sold royalty-free for $3,500. Prices include site development licenses and six months of free support and updates. smx products may be purchased on a 30-day trial basis.

For more information contact Micro Digital Inc., 6402 Tulagi St., Cypress, CA 90630-5630. 800-366-2491 or +1-714-373-6862. FAX: +1-714-891-2363.

DataViews Announces DV-Centro 1.2

DataViews Corporation has announced DV-Centro 1.2, a development tool for building Visual Programming Languages (VPLs) under Windows NT. DV-Centro allows programmers to create true VPL systems based on custom symbols, shapes, and objects connected to data. Based on an object-oriented C++ framework, DV-Centro features a suite of graphical tools that allow users to create reusable components for complex VPL applications. Also featured is an Interactive Control Engine that manages events between the user display and the application components.

DV-Centro 1.2 features a set of coordinated C++ classes optimized for managing the relationships between graphics and data in VPL applications. This framework decreases the amount of code that has to be programmed, tested, and debugged. DV-Centro also includes a set of C++ utilities for memory management, infinite undo/redo, object storage/retrieval, and journaling.

DV-Centro 1.2 supports Windows NT 3.51 with Microsoft's Visual C++ compiler, HP-UX 10.x, and Sun Solaris 2.4 with a native C++ compiler. Development seats for DV-Centro begin at $15,000.

For more information contact DataViews Corporation, 47 Pleasant St., Northampton, MA 01060. +1-413-586-4144. FAX: +1-413-586-3805. e-mail: info@dvcorp.com. WWW: http://www.dvcorp.com

Rational Announces Rational Rose for Java

Rational Software Corporation has announced Rational Rose for Java, providing developers the capabilities for object modeling, controlled iterative development, and round-trip engineering. In addition, Rational released a Java-ready extension of the Rational Apex C/C++ environment, enabling software managers to establish and maintain control of their application architecture and providing integration of Java with C and C++ code. Together, the two products offer support for large-scale Java development efforts.

Rational Rose/Java starts at $2,400 (U.S. price) for the Windows version, which includes the first year of support. Rational Apex C/C++ pricing ranges from $7,500 to $12,500, depending on configuration and bundling options.

For more information contact Rational Software Corporation, 2800 San Tomas Expressway, Santa Clara, CA 95051-0951. 800-728-1212 or +1-408-496-3600. FAX: +1-408-496-3636. e-mail: product_info@rational.com. WWW: http://www.rational.com.

REQUISITE Announces RequisitePro v1.0

REQUISITE, Inc. has announced RequisitePro 1.0, a team-based requirements management tool that integrates traceability and the version control product, PVCS Version Manager. The Windows-based tool provides the ability to follow traceability relationships through a structured hierarchy. For example, when a high-level user needs a change, a developer can quickly identify which software elements must be altered. A tester can pinpoint which test protocols must be revised. Managers can better determine the potential costs and difficulties of implementing the changes. In addition, RequisitePro builds a requirements database that is attached and synchronized to the projects specifications and test plans. This repository holds a development team's documents, requirements, and traceability relationships.

With RequisitePro, individual requirements are itemized from within Microsoft Word project documents and managed with an integrated database. All information is centralized in one shared repository. Requirements documentation, manage-

ment, traceability, and reporting are now performed with one tool.

For more information contact REQUISITE, Inc., 4720 Table Mesa Dr., Ste. F, Boulder, CO 80303. 800-732-4047 or +1-303-499-9177. FAX: +1-303-499-9535. e-mail: 74067.612@compuserve.com.

❑

NetManage Ships Swift 2.0 IntraNet Access Package

NetManage, Inc. has released Swift 2.0 and Swift IntraNet Package 2.0. Swift provides company-wide host access services over multiple protocols for Windows 3.1, Windows 95, and Windows NT. Based on the host access components of Chameleon IntraNet Desktop suite, Swift 2.0 supports access to IBM mainframes, AS/400 and VMS/UNIX hosts over TCP/IP, SNA, and asynchronous connections. It includes host emulation services, NetManages' WebSurfer HTML browser, industry-standard SMTP e-mail, and the NS/Router client software.

Swift 2.0 costs $199. The Swift IntraNet Package is available in a minimum quantity purchase of 20 copies for $420 per copy.

For more information contact NetManage, Inc., 10725 North De Anza Blvd., Cupertino, CA 95014. +1-408-973-7171. FAX: +1-408-257-8780. e-mail: marketing@netmanage.com. WWW: http://www.netmanage.com.

❑

QSI Ships ProTEST 3.5 for Windows

QSI has begun shipping ProTEST 3.5, a test process management system designed to assist in the development of a formal and structured test methodology, and manage the testing process. Enhancements include the ability to assign weight factors to a test case procedure, test case summary reports that provide detailed status of the test case, and an online "Regression Testing - To Do" file that allows direct access to the test cases that have failed, in order to include them as part of the Regression Test Phase of the project. Additional enhancements include test case status update capabilities that allow testers to pick and choose which test cases to update with a Pass/Fail or Incomplete status. Another enhancement is a Quick View function that provides a display of all the test case procedures in one screen, along with the associated requirements, problem report numbers, description of the procedures, and the status of each individual test case procedure.

For more information contact Quality Systems International, Inc., 416-420 Highland Ave., Cheshire, CT 06410. 800-557-9787 or +1-203-699-9787. FAX: +1-203-699-9789.

❑

ObjectSpace Announces Web<ToolKit>

ObjectSpace, Inc. has announced Web<ToolKit>, an ANSI/ISO-compatible C++ class library for producing HTML World Wide Web pages. Web<ToolKit> supports page creation using a set of C++ classes representing HTML elements, including text, links, graphics, tables, forms, frames, and widgets.

Web<ToolKit> uses ANSI/ISO standard containers and strings. Version 1.0 operates with existing ObjectSpace products, including STL<ToolKit> and Systems<ToolKit>. The product can also be used with Borland 5 STL and the GNU STL.

Cost for the software is $349 for PC platforms and $475 for UNIX platforms. Special bundled pricing is available with STL<Toolkit>, ObjectSpace's implementation of the Standard Template Library, and Systems<ToolKit>, ObjectSpace's second generation C++ toolkit.

For more information contact ObjectSpace, Inc., 14881 Quorum Dr., Ste. 400, Dallas, TX 75240 +1-214-934-2496. FAX: +1-214-663-9099. e-mail: training@objectspace.com. WWW: http://www.objectspace.com.

❑

Programmer's Paradise Releases Communications Paradise Catalog

Programmer's Paradise, Inc. has released a new catalog featuring products for Internet and intranet development, communications, and connectivity. Published quarterly, the Communications Paradise Catalog features HTML browsers, Java applet creation tools, web servers, authoring and editing tools, search engines, and an array of communications and connectivity products including modems and peripherals, navigators, TCP/IP, and e-mail software.

For more information contact Programmer's Paradise, Inc., 1163 Shrewsbury Ave., Shrewsbury, NJ 07702-4321. +1-908-389-8950. FAX: +1-908-389-9227.

❑

IBM Expands VisualAge for C++ to Cover Windows

IBM has added support for Windows NT and Windows 95 to VisualAge for C++. VisualAge for C++ for Windows provides an integrated C++ development environment that includes a data access builder, which automatically generates objects and parts for accessing relational data.

The data access builder has been enhanced in the VisualAge for C++ for Windows release to support Open Database Connectivity (ODBC). In addition to native support for DB2 for Windows NT, this builder can access Oracle, Sybase, and other relational databases supported by ODBC.

VisualAge for C++ for Windows also includes an optimizing compiler that supports the Pentium ProChip, a syntax-highlighting editor, class browser, program debugger, and performance analyzer.

VisualAge for C++ for Windows v3.5 costs $449. Customers using any competitive product with a C++ compiler on Windows will be offered a special upgrade base price of $299.

For more information contact International Business Machine Corporation, Route 100, P.O. Box 100, Somers, NY 10589. 800-426-3333. WWW: http://www.software.ibm.com/info/ \ pr0305.html.

❑

Micro Focus Ships Revolve Year 2000 Add-On

Micro Focus has announced the shipment of the Micro Focus Revolve Year 2000 Add-On and Challenge 2000 packages. The Revolve Year 2000 Add-On is one component of Micro Focus Challenge 2000 v2.0, a product that includes tools and services for addressing all phases of the millennium date-change problem, from assessment and identification through implementation and testing.

The Revolve Year 2000 Add-On tracks the analysis and changes required for Year

Mainsoft Ships MainWin Test 3.0

Mainsoft Corporation has begun shipping MainWin Test 3.0, an automated testing tool compatible with Microsoft Test 3.0 for testing Windows applications ported to UNIX platforms. MainWin Test 3.0 offers developers a testing library featuring tests for the latest Windows controls; powerful and easy-to-learn TestBasic scripting language; an integrated development environment with visual syntax editor, compiler, and debugger; and screen capture/compare and dialog capture/compare utilities.

MainWin Test for UNIX offers compatibility with Microsoft Test for the PC, both in test-script development and execution. MainWin Test allows the reuse of the same testing scripts created for validating the Windows product.

MainWin Test will be supported on DEC, HP, IBM, SCO, SGI, and Sun UNIX. Developer pricing for MainWin Test starts at $5,000.

For more information contact Mainsoft Corporation, 1270 Oakmead Parkway, Ste. 310, Sunnyvale, CA 94086. +1-408-774-3400. FAX: +1-408-774-3404. e-mail: info@mainsoft.com. WWW: http://www.mainsoft.com.

2000 compliance using assessment, analysis, and modification features. The Year 2000 Add-On enables users to examine, visualize, and assess systems consisting of thousands of programs and millions of lines of code.

Micro Focus is also introducing two customized packages designed to match the needs of organizations addressing their Year 2000 problem. These packages, the Challenge 2000 Pilot and the Challenge 2000 Compile and Test Option, provide a Year 2000 team with the tools, training, and support to launch their assessment, analysis, and modification efforts. The Challenge 2000 Pilot costs $46,500. The Challenge 2000 Compile and Test Option pricing starts at $43,000.

For more information contact Micro Focus, 2465 East Bayshore Rd., Palo Alto, CA 94303. +1-415-856-4161. FAX: +1-415-856-6134. WWW: http://www.microfocus.com.

PLATINUM Releases Paradigm Plus v8.0

PLATINUM technology, inc. has released Paradigm Plus v8.0, an object-oriented analysis and design tool that supports version 0.8 of the Booch/Rumbaugh/Jacobson Unified Modeling Language (UML), Enterprise Component Modeling (ECM), and round-trip engineering. With Paradigm Plus, companies create "components" that divide information systems into manageable sections. Components are stored in the Paradigm Plus object repository where they are organized and synchronized with source code and documentation.

Paradigm Plus generates code in C++, Ada, Smalltalk, PowerBuilder, Java, ObjectPro, and Forte, as well as RDBMS and ODBMS schema definitions. It also reverse engineers C++, Smalltalk, PowerBuilder, ObjectPro, and Forte code. The combined forward and reverse engineering capabilities of Paradigm Plus, without any loss of data, provide round-trip engineering, which enables iterative development. Utilizing a distributed object repository and open architecture, Paradigm Plus optimizes reuse by allowing access to all stored objects and relationships in a multi-user environment. Paradigm Plus components also can be published on the Web as HTML documents, where they can be viewed, downloaded, and reused.

Paradigm Plus 3.01 is available for Windows 3.1, Windows NT, and Sun Solaris.

For more information contact PLATINUM technology, inc., 1815 South Meyers Rd., Oakbrook Terrace, IL 60181-5241. 800-442-6861 or +1-708-620-5000. FAX: +1-708-691-0710. e-mail: info@platinum.com. WWW: http://www.platinum.com.

JBA Introduces JBA Guidelines v3.3

JBA International has introduced JBA Guidelines v3.3, a software development environment for OS/2 WARP and OS/2 v2.1, for generating GUI-based client/server applications. Guidelines has incorporated a messaging layer that ensures that all applications built today are object-enabled. Version 3.3 enhances the distributed message layer by providing an interface between standardized object calls and the code servicing these calls. This code can be a mixture of existing logic and more advanced CORBA-compliant objects.

Currently Guidelines is packaged in two versions: Guidelines for Desktop Developers, with prices starting at $595, and Guidelines for Corporate Developers, priced at $7750 per seat. A Base Pack, in essence a Guidelines tester, is free and can be downloaded from CompuServe (GO Guidelines) or the Internet (http://wwwjba.co.uk).

For more information contact JBA International, 161 Gaither Dr., Ste. 200, Mt. Laurel, NJ 08054. 800-522-4685 ext. 3074. FAX: +1-609-222-1952. WWW: http://www.jba.co.uk.

Art Technology Group Introduces Dynamo 2

Art Technology Group has introduced Dynamo 2, a Java-based, Internet application engine. Dynamo 2 enables Web developers to prototype, build, and deploy scaleable, high-volume, and database-driven Web applications requiring dynamic page generation and personalized content. Based on open standards, Dynamo 2 allows Web developers to embed native Java code (not Javascript) into HTML files to extend functionality on the server side. Java code is compiled on the server to customize everything from the visual characteristics of the page to behaviors of traditional client-side Java applets.

Pricing for Dynamo 2 is $5,000 per license, and one year of support can be purchased at $750/year per license. Support is available via the Web, where users can get up-to-date technical information, receive technical support, and download new "Gears" and releases.

For more information contact Art Technology Group, +1-617-859-1212. e-mail: info@atg_dynamo.com. WWW: http://www.atg-dynamo.com.

Letters to the editor may be sent via email to cujed@mfi.com, or via the postal service to Letters to the Editor, C/C++ Users Journal, 1601 W. 23rd St., Ste 200, Lawrence, KS 66046-2700.

Dear Dr. Plauger,

I am sending you the following suggestion in the hope that you can communicate it to the ANSI C++ committee. I realize they are close to freezing the language, and may already have done so by now, but still I thought it was worth a try.

A widely acknowledged shortcoming of C++ is the visibility (in header files) of private attributes and methods. When a new private method is added, files that include the header only for the public interface are needlessly recompiled. This can become a major nuisance in complex projects with many interdependent files.

I propose a simple addition to the language to fix this. It would not affect existing code. The idea is to allow private member functions to be defined in the implementation file without being declared in the class declaration. Such functions would have file scope, just like static functions commonly used in C. Unlike static functions in C, however, they would have access to all the private attributes and methods of the class to which they belong. The syntax could use the keyword `static`, thus:

```
static void MyClass::
    method_with_file_scope( void ) {
    // function defined here
    }
```

The compiler would disallow the declaration of the same function in the header file. If one more meaning of the already overloaded keyword `static` is objectionable, the keyword `private` (or some new keyword) could be used instead. Like conventional private member functions, these functions would not be accessible to a derived class. They would not be accessible to a friend class unless the friend is implemented in the same file.

In my experience, private member functions change more often than data members and other functions, especially during development, and they usually all live in the same source file. Although this simple change would not eliminate all needless recompilations, it would surely help.

Sincerely yours,

Jack Wathey

Dear P.J. Plauger,

Declaration of non-virtual private member functions or, did Bjarne nod? Having been a C programmer for many years, I was looking forward to the opportunity of doing some serious work in C++. But I soon found vexation and frustration.

I'm used to hiding the implementation for an "object" in C by declaring what you need to know to use it in a header file, and the rest in a private header file or in a .c source file. But it turns out that because member functions have to be declared in the header, a lot of implementation shows. I'm talking about the sort of functions you would usually declare as static. Like

```
for (i = 0; i < something ; i++) {
    dosomethingto(fred[i]) ;
}
```

where `dosomethingto` is a strictly local function invented on the spur of the moment. But in C++, you have to declare it in the header. If I try this, my current compiler says something like "`dosomethingto` is not a member function of class `C`." This means that you have to advertise your implementation to all your users in the header file.

In the ARM, this requirement is justified to prevent people from "hijacking" the class, like this:

```
class C {
private:
    int t;
...
}
void C::hijack() { t++ ; }
```

But I think that this goes against Stroustrup's admirable attitude generally, that protection is there to prevent you inadvertently doing something you didn't mean to do, and not to stop you from doing what you want to do. If you wanted to hijack the class even more thoroughly you could write one or all of the member functions yourself. With most compiler/linker combinations, you would get your own implementation and no warning that there was another in the library. But this isn't called "hijacking," it's called "implementing a class!"

My proposition is that a declaration/definition of a member function that is not declared in the class should be accepted, it should be deemed to be private, its name should have file scope. Being private it would only be able to be called from other member functions (that takes care of the hijacking issue). One good thing about this is that it is not an addition to the language, and it doesn't break any existing correct code. Such functions don't affect the size of the members of the class, so it's feasible (as far as I can see).

Maybe this idea has already been discussed, and maybe even proposed. Maybe it's even on its way! Meanwhile, I have adopted another approach for my own implementation, which would go like this for class `C`:

```
class C_imp;      // implementation of C
class C {
    friend class C_imp;
private:
    C_imp * cp ;
public:
    // public functions of C
    ...
protected:
    // functions derived classes can
    // inherit
    ...
};
```

So all the private data and functions of this class can be declared in my private header file `<c_imp>`

The constructor for `C` has to do a `new` on the `cp` member — the only private member. The functions of `C` then provide a front for the real functions in `C_imp`. It also hides what possible classes I may or may not have inherited from in building my class.

I agree that this is a bit of a kludge, but I think, for a class of any size, it's pretty important to hide the implementation. I still have to stick declarations for all my tiny functions into the (local) declaration of `C_imp`, but I don't have to burden the customer with them. What's more, when they get a re-release of the code, they don't have to recompile, but only re-link.

I would be interested to know what you think about this proposal.

Yours,

Dan Oestreicher

The scheme you outline is a technique sometimes used to provide additional hiding of implementation details. Even it can be hijacked, by someone who knows, or can guess, enough of those hidden details. "Private" doesn't necessarily mean "secret." It is more a way to minimize accidental use of internal information about a class. I know of no proposals to add the kind of extension that both you and Mr. Wathey suggest, and I doubt they'll be forthcoming at this late stage. — pjp

Dear Mr. Plauger,

I am working for a software company which has the majority of the products written in C. We think that we should shift to C++ where possible. We are trying to gather information about how it can be done from those who experienced the transition. What is a good way to start? Is there a formal procedure? Where can we get information about this? Any advice is appreciated. Thanks.

Mustafa Yorgancioglu
Tempe, AZ

A number of articles have been written in the trade press, over the past few years, describing experiences in moving from C to C++. You will find a few in back issues of CUJ. *In fact, Bobby Schmidt's column, "The Learning C/C++urve," focuses especially on how to migrate code from C to C++. The basic rule is to take small steps, preserving old code in a mixed environment as much as possible. C++ works best for writing new code designed from the outset with object-oriented design principles in mind. — pjp*

Dear Mr. Plauger:

You might as well do it. Get it over with. Do us all a favor. Please go ahead and change the name of your once fine magazine to *The C++ Users Journal* so it accurately reflects the content within, and so I can no longer accuse you of false advertising. I for one still refuse to give up to the legions of bloated C++ programmers spawned by Mr. Gates and his like. Long live C for embedded programming...

Steadfastly resisting the hype,

Mark S. Outson

I think it's the programs that are bloated, not the programmers, and I have trouble blaming Bill Gates for C++. Still, we have been running light on "strictly C" articles in recent months. You'll be happy to know we've taken steps to acquire more C articles. Since C underlies so much of C++, a good C article can be relevant to all programmers — it doesn't matter whether your forte is C or C++. Of course, as the next letter demonstrates, not everyone shares your perception of our content.— pjp

---

Dear Mr. Plauger,

Alas, I find I must bid a fond farewell to *The Journal,* and it does seem fair, considering the many letters you receive extolling the virtues of good old *CUJ,* that I should address this to you. After all, balance is important. Right?

I, unfortunately, am one of those who've found that with each passing issue less and less of any worth (program-wise) appears in The Journal. Come to think of it, *CUJ* has never been long on programming information.

Lots of coding, to be sure, but except for an occasional article on how to handle this or that side-saddle piece of equipment, or how to rework a few obscure statements in an algorithm of rare vintage and even rarer utility, the overriding gist of what's published in *The Journal* (its one claim to fame) has always been:

> We peddle esoterica and are loudly proud of our total committment to the protection and preservation of the purity of the cryptic qualities of the work performed by ANSI C Standards Committee XYZ789.

Hey, isn't that how Gurus earn a living?

Truly,

Mark Rhyner

I personally earn most of my income writing code in C and C++ that people find worth licensing. But we can't please everyone with this magazine. Sorry to see you go. — pjp

---

Dear Mr. Plauger,

I am a novice programmer in C. The past two months I've been trying to teach myself the language. The instrument that I've been using

has been the Waite Group's beginning primer book, *The New C Primer Plus.* My question to you is the following: Are the articles presented in your periodical *C/C++ Users Journal* suitable for an individual with my skill level in C programming? Secondly, if my skills are not adequate what materials can you suggest that I investigate and acquire to raise my skill level adequately so as to take advantage of the material presented with in *The C/C++ Users Journal?*

Any advice or encouragement would be greatly appreciated.

Thank you for your time and attention.

David Hodge
dlhodge@netcom.com

I suspect that only a small fraction of our articles will make sense to you right now, but that should change rapidly as you gain experience with C. The only way I know to learn any programming language is to practice writing code, preferably useful code. The more you write, the more you learn. Good luck. — pjp

---

Hi Mr. Plauger,

A few months ago I started programming in C and Visual C++ again. I used C in some of my number crunching programs a few years ago. I am currently working on an embedded control system on the iRMX OS and multibus II arch (a real-time program).

I came across your magazine a few months ago...very interesting stuff. I purchased the May'96 and several of the '95 publications, all that I could find. I am in the process of ordering the CD.

Question: Has your magazine ever published any articles in the following areas: database and MPEG?

BTW I was going through the Q&A in one of the '95 pubs and wow...I was able to solve one of my lvalue/rvalue errors...many thanks!! Keep up the good work.

Cheers

Anand

The May 1996 issue which you purchased includes a database-intensive article, "Object Persistence with Relational Databases," by Chuck Allison. Another recent database-related article is "C Database Programming with ODBC," by Alex Ragen, in the November 1995 issue. We have yet to run an article on MPEG, but we would not rule it out as a future topic.

We're glad you found Pete Becker's Q&A column so immediately and tangibly useful. I imagine many of our readers have had the same experience. Of course, if you want to be sure you don't miss out on anything in the future, you could always suscribe by sending an e-mail to `rdorders@mfi.com` *(hint, hint). Thanks for reading* CUJ *and thanks for writing. — mb*

Dear Mr. Plauger

I would like to open with a compliment on your articles in *C/C++ Users Journal*; they are both informative and well written.

I have been seeking a solution to a C++ problem for several weeks now and am no closer. I had thought to write to you earlier but did not wish to waste your time on a simple problem. For me it no longer seems simple.

I have been writing a database library for my own interest and exercise over the last several weeks. What interested me about this topic was the varied number and composition of fields that could be encountered at run time. After iterating through many data types I settled on an array of void pointers that I would cast with (CAST) or DYNAMIC_CAST<> at run time.

Here is a code sample of how I set it up:

```
void **fields;
for (i=0;i<NumberOfFields;i++)
    switch(FieldType[i]){
        case 'C':
            ((void) (field))[i] =
                new string;
        case 'N':
            ((void) (field))[i] =
                new int;
...
```

The compiler does not know how to increment when cast to void. (This is similar to Smalltalk's bag class). Since I know the order and the type of each field I could enjoy the convenience of accessing them as an array. The problem is it is an array composed of hetrogeneous types.

I had also considered unions and a generic class. The union presented two problems: first, it wasted space; second, I wished to access the fields by the same name.

Another thought was a generic class that held all fields as characters. I felt this required too much conversion and I also would need a single function that returned different types depending on the type that the character sequence represented.

In short, I am looking for an array of pointers that allow me to access these elements, regarless of type, by the same name (ie. field[i].value).

I hope I have been clear here. Thank you very much for your time.

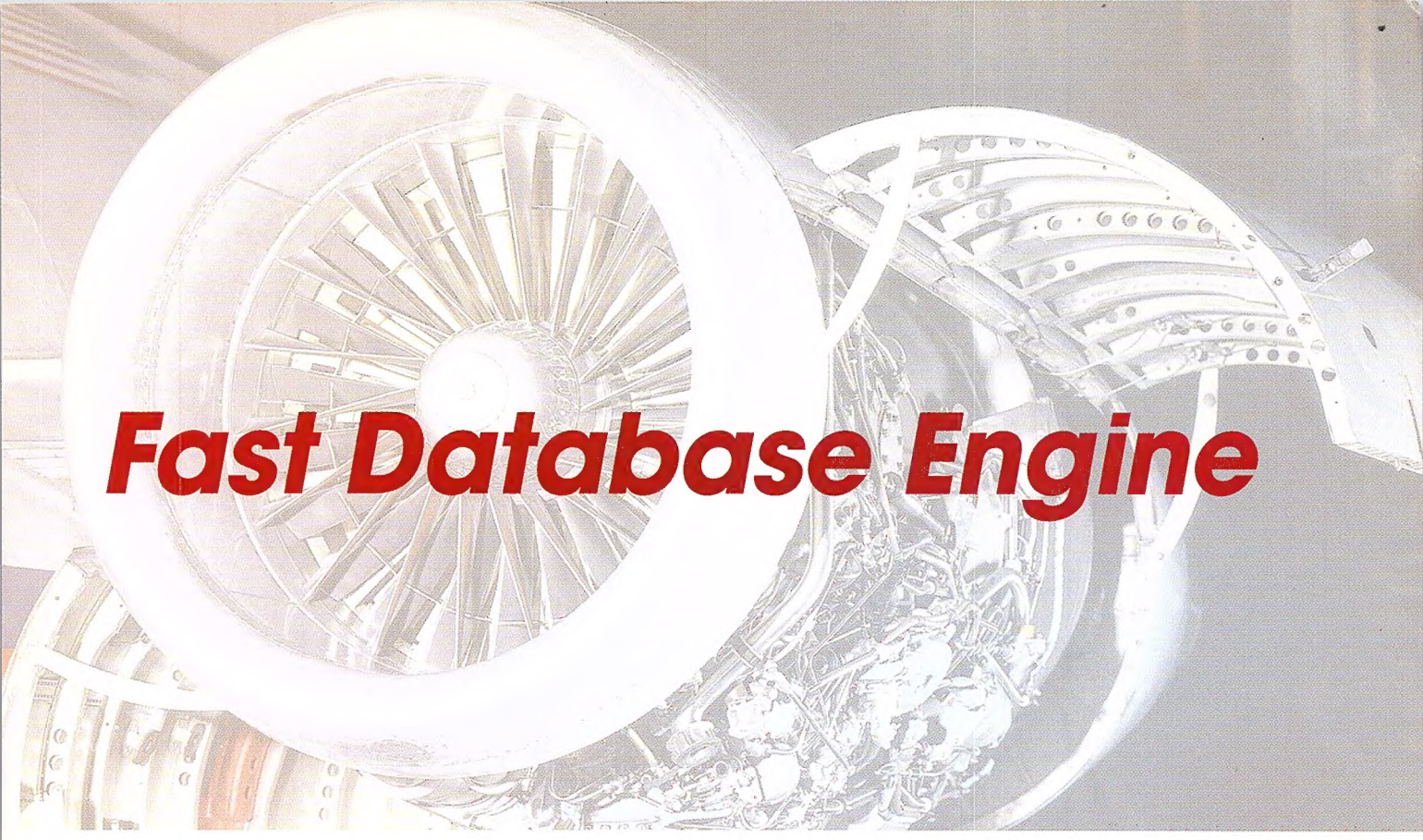
Sincerely,

Phill Moran

You're pretty close. Just define typenames for the various field types, then cast the void pointers appropriately, as in:

```
void **fields;
for (i=0;i<NumberOfFields;i++)
    switch(FieldType[i]){
        case 'C':
        *((Field_type_C *)field[i])
           = new string;
        case 'N':
        *((Field_type_N *)field[i])
           = new string;
...
```

— pjp



Fast Database Engine

Now discover CodeBase 6.1 – The fastest and smallest database engine available with xBASE file compatibility!

Query a million records in just 0.97 of a second, or add ten thousand records in just 2.17 seconds.

Discover these valuable benefits of CodeBase:

- C, C++, Visual Basic and Delphi are all supported.
- FoxPro, Clipper and dBASE file compatible, and multi-user compatible
- Runs under Windows 95, 3.1, NT, DOS, and UNIX
- Tiny DLL loads quick and takes little memory
- Transaction processing and logging
- Client/Server configuration included
- Crash recovery is built-in
- Full featured report writer included
- Source code included
- Royalty Free–even with Client/Server

SQL Option Available

Now get the most database power at the lowest cost.

★ 1991 ★ DATA BASED ADVISOR Readers Choice Award
★ 1992 ★ DATA BASED ADVISOR Readers Choice Award
★ 1993 ★ DATA BASED ADVISOR Readers Choice Award
★ 1994 ★ DATA BASED ADVISOR Readers Choice Award
★ 1995 ★ DATA BASED ADVISOR Readers Choice Award

Award-Winning performance 5 years in a row!

"CodeBase gives ACT! the fast database access contact management users need"

- Michael Plasterer, Director of Development, Symantec

The Database Engine for Programmers

FREE 30 Day Test Drive

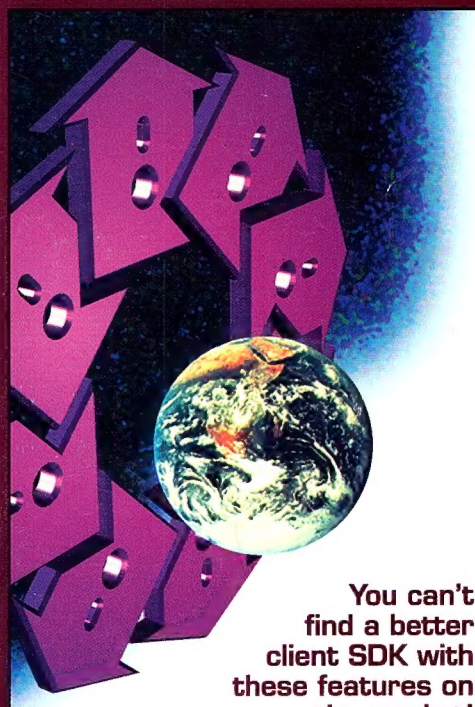
Test drive the new CodeBase 6.1 for 30 days with your own code. No risk. No obligation. Order yours today.

Call: 403-437-2410

SEQUITER SOFTWARE INC.
Fax: 403-436-2999
Email: info@sequiter.com
P.O. Box 575 Newmarket NH 03857-0575
Web site: www.sequiter.com

❑ Request Reader Service #101 ❑

PLATFORMS: WINDOWS NT • WINDOWS 95 • OS/2 • NLM • DOS • APPLE A/UX • LYNX • SCO • SYSTEM MANAGER • SUN SPARC-SOLARIS • IBM RS/6000 • APPLE SYSTEM 7 • AT&T UNIX • BANYAN • INTERACTIVE UNIX • LINUX • MOTOROLA 88OPEN • QNX • HP9000 • RS/6000 • SUN O/S 4.X • SUN O/S 5.X • MIPS ABI (SGI)



Q: What does it take to deploy a superior client/server application?

A: *A SUPERIOR SERVER*

START with the most advanced client-side SDK on the market: c-tree® Plus at $895.

- Complete "C" Source code
- ROYALTY FREE (Client Side)
- Multiple supported protocols
- Fast, portable, reliable
- Powerful features like transaction processing
- Win95, NT, and Windows 3.1 ready

ADD a strong, multi-platform, industrial-strength Server that supports.

- File mirroring
- Heterogeneous networking
- Automatic disaster recovery
- Multi-threaded design
- Best price/performance available: from $445- $3745

RESULT? A solid, economical, easily deployable product that fits your needs.

- Portable
- Scalable
- Exceptional Performance
- Flexible
- Easy Server distribution
- Convenient OEM terms

You can't find a better client SDK with these features on the market!

Over sixteen years of proven reliability and performance.

No one else supports over 30 platforms in this price range!

c-tree Plus®

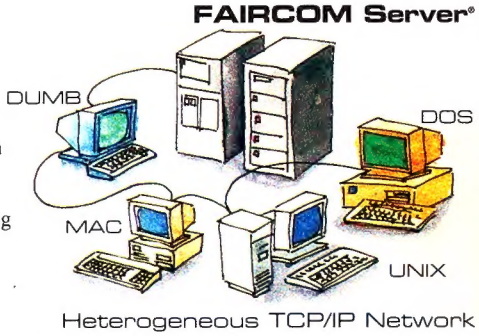
After 16 years, still the best in single-user, multi-user and client/server development. Now available for Windows 3.1, Win95, and Windows NT. c-tree Plus is still distributed in complete C source code and is known for its unparalleled flexibility, portability and royalty-free licensing policy. This licensing puts your development budget to work for you.

- Complete C Source
- Single/Multi User
- Client/Server (optional)
- Full ISAM functionality
- No Royalties
- Transaction Processing
- Fixed/Variable Length Records
- High Speed Data/Index Caching
- Batch Operations
- File Mirroring
- Multiple Contexts
- Unsurpassed Portability

FairCom Server®

Many major companies have turned to FairCom to implement their client/server systems. FairCom Servers adhere to the latest client/server design principles. FairCom Servers communicate using NETBIOS, SPX, TCP/IP, AppleTalk, shared memory, Message Queues or StreetTalk, depending upon the server platform. Most server platforms support more than one protocol, providing you the flexibility to choose how your client applications communicate to the server. For example, the FairCom OS/2 Server can communicate with shared memory, NETBIOS, SPX, and TCP/IP clients at the same time. No one else offers these features and portability for the price:

- Client/Server Model
- Transaction Processing
- Requires <2MB RAM
- Online Backup
- Disaster Recovery
- Rollback - Forward
- Anti-Deadlock Resolution
- Client-side "C" Source
- Multi-threading
- Heterogeneous networking
- File Mirroring
- OEM/Source Available



FOR YOUR NEXT PROJECT CALL FAIRCOM: YOU CAN'T FIND A BETTER HETEROGENEOUS CLIENT/SERVER SOLUTION!

FAIRCOM® CORPORATION

Also inquire about these FairCom products:
d-tree™ r-tree© ODBC Driver

WWWeb Address: http://www.faircom.com/

U.S.A. 4006 W. Broadway - Columbia, MO 65203-0100 • phone (573) 445-6833 fax (573) 445-9698
EUROPE Via Patrioti, 6-24021 Albino (BG) - ITALY • phone (035) 773-464 fax (035) 773-806
JAPAN IKEDA Bldg. #3,4f-112-5, Komei-chou - Tsu-city,MIE 514 Japan • phone (0592) 29-7504 fax (0592) 24-9723

(800)234-8180